

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Rok Bertoncelj

Parsek CMS 2.0

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Marko Bajec

Ljubljana, 2010



Št. naloge: 01638/2010

Datum: 15.03.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ROK BERTONCELJ**

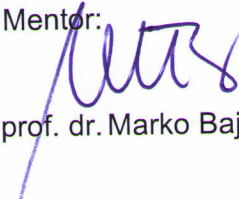
Naslov: **PARSEK CMS 2.0**
PARSEK CMS 2.0

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

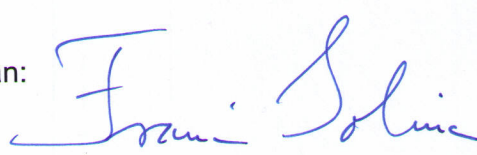
V okviru diplomskega dela razvijte sistem za upravljanje z vsebinami (ang. CMS – Content Management System), ki naj bo kar se da splošen, da ga bo moč uporabljati za različne namene. Pri razvoju sistema sledite uveljavljenim metodologijam. Najprej pregledajte, kakšna orodja so na voljo na trgu in se prepričajte, da tovrstnih splošno namenskih odprtokodnih orodij ni na voljo. Za tem izdelajte najprej koncept rešitve, pripravite načrt ter sistem izdelajte. Vse faze razvoja ustrezno dokumentirajte.

Mentor:


prof. dr. Marko Bajec



Dekan:


prof. dr. Franc Solina

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si



Št. naloge: 01638/2010

Datum: 15.03.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ROK BERTONCELJ**

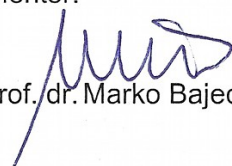
Naslov: **PARSEK CMS 2.0**
PARSEK CMS 2.0

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

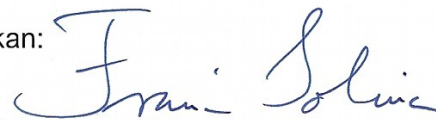
V okviru diplomskega dela razvijte sistem za upravljanje z vsebinami (ang. CMS – Content Management System), ki naj bo kar se da splošen, da ga bo moč uporabljati za različne namene. Pri razvoju sistema sledite uveljavljenim metodologijam. Najprej preglejte, kakšna orodja so na voljo na trgu in se prepričajte, da tovrstnih splošno namenskih odprtokodnih orodij ni na voljo. Za tem izdelajte najprej koncept rešitve, pripravite načrt ter sistem izdelajte. Vse faze razvoja ustrezno dokumentirajte.

Mentor:


prof. dr. Marko Bajec



Dekan:


prof. dr. Franc Solina

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Rok Bertoncelj,

z vpisno številko 63040010,

sem avtor diplomskega dela z naslovom:

Parsek CMS 2.0

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom prof. dr. Marka Bajca
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 13.09.2010

Podpis avtorja:

Zahvala

Za pomoč pri izdelavi diplomske naloge se zahvaljujem mentorju, prof. dr. Marku Bajcu in sodelavcema, Tomažu Cerarju ter Andreju Rakovcu, za koristne nasvete in ideje.

Kazalo

1. Uvod.....	1
2. Koncept rešitve.....	4
2.1 Arhitektura.....	4
2.2 Drevesna kategorizacija vsebin.....	4
2.3 Odjemalec in uporabniški vmesnik.....	6
2.3.1 Arhitektura.....	6
2.3.2 Uporabniški vmesnik in funkcionalnosti.....	7
2.3.3 Izbira tehnologije.....	9
2.4 Strežnik.....	11
2.4.1 Izbira tehnologij.....	12
2.5 Meta opisi in predstavitev vrednosti atributov.....	14
2.5.1 MetaAsset.....	14
2.5.2 MetaField.....	15
2.5.3 Asset.....	15
2.5.4 Primer.....	16
2.6 Standardni tipi vsebin.....	17
3. Realizacija in delovanje.....	19
3.1 Inicializacija uporabniške seje.....	19
3.1.1 Analiza entitetnih razredov.....	19
3.2 Pretvorba entitete v Asset.....	21
3.3 Generiranje urejevalnika.....	23
4. Konfiguracija in namestitev.....	24
4.1 Dodatna konfiguracija z anotacijami.....	24
4.1.1 CMSAsset.....	24
4.1.2 CMSField.....	25
4.1.3 TitleField.....	25
4.2 Namestitev.....	26
4.2.1 Skupaj s ciljno aplikacijo.....	26
4.2.2 Samostojna namestitev.....	27
5. Obstoječi sistemi.....	28
5.1 Parsek CMC.....	29
5.1.1 Tehnologija Parsek CP2.....	29
5.1.2 Uporabniški vmesnik.....	29
5.1.3 Še nekatere ključne lastnosti.....	30
5.1.4 Primerjava.....	33
5.1.5 Možnost migracije na Parsek CMS 2.0.....	33
5.2 Magnolia CMS.....	33
5.2.1 Content repository.....	34
5.2.2 Urejanje vsebin.....	35
5.2.3 Standard Templating Kit.....	37

5.2.4 Modul Data in urejanje poslovnih objektov.....	39
5.2.5 Primerjava s Parsek CMS 2.0.....	39
5.3 Metawidget.....	40
6.Zaključne ugotovitve.....	44
Priloga A: Entiteta Message.....	45
Priloga B: Meta opisi in Asset.....	47
1. MetaAsset.....	47
2. MetaField.....	48
3. MapMetaField.....	49
4. Asset (instanca vsebine).....	49
Priloga C: MetaDataFactory algoritmi.....	52
1. createMetaAsset.....	52
2. createMetaField.....	53
3. Pomožne metode.....	57
Priloga D: pomembnejše metode razreda Core.....	59
Priloga E: pretvorba entitete v Asset.....	61
Priloga F: Opisi storitev (services).....	64
CmcService.....	64
EditorService.....	64
GridService.....	65
ProductEditorService.....	67
SearchService.....	67
TreeService.....	68
Priloga G: Podrobnosti odjemalca.....	70
1. Uporabniški vmesnik.....	70
2. Generiranje vnosnih polj.....	73
2.1 Preprosta vnosna polja.....	73
2.2 Kompleksna vnosna polja.....	74
2.3 Prilagojeni urejevalniki.....	75
3. Iskanje.....	77
Priloga H: Razredni diagrami.....	80
Priloga I: Seznam slik.....	81
Viri.....	82

Seznam uporabljenih kratic in simbolov

API – application programming interface
CASE – Computer-aided software engineering
CMS – Content Management System
CSS – cascading style sheets
EJB-QL - Enterprise JavaBeansQuery Language
GWT – Google Web Toolkit
GXT – Ext GWT
HTML – hyper text markup language
Java EE – Java Enterprise Edition
JPA – Java Persistence API
JSF – JavaServer Faces
ORM – Object-relation mapping
Parsek CMC – Parsek Content Management Console
Parsek CP2 – Parsek Content Platform 2
POJO – plain old Java object
RPC – remote procedure call
SQL – structured query language
SUPB – sistem za upravljanje podatkovne baze
UUID – universally unique identifier
XML – extensible markup language

Povzetek

Diplomsko delo opisuje sistem za upravljanje z vsebinami (CMS – Content Management System), ki je bil zasnovan in razvit za urejanje vsebin oziroma poslovnih objektov v spletnih aplikacijah. Njegova glavna lastnost je, da vnaprej ne ve kakšne entitetne tipe se bo urejalo in zato njihove lastnosti samodejno razbere ter na podlagi rezultatov analize samodejno zgenerira ustrezen urejevalnik. To omogoča enostavno uvedbo na različne projekte s kar se da malo prilagajanja.

Najprej je podan koncept rešitve, kjer so predstavljene osnovne ideje, arhitektura in funkcionalnosti sistema, nato pa so konkretnije opisani nekateri pomembnejši postopki od analize entitetnih tipov do samodejnega generiranja urejevalnika. Nazadnje so predstavljene še možnosti konfiguracije in postopek namestitve sistema.

V zaključku je podan opis nekaterih obstoječih sistemov in primerjava s Parsek CMS 2.0, na koncu pa je ovrednoten rezultat dela, podane pa so tudi nekatere misli v zvezi z bodočim razvojem sistema.

Ključne besede:

Spletna aplikacija, sistem za urejanje vsebine, Java EE, ORM, JPA, GWT, meta programiranje.

Abstract

The thesis describes a content management system (CMS) that was designed and developed for management of bussiness objects in web applications. It can automatically parse the properties of entities from the underlaying data model and the client can dynamically generate the appropriate editor for each entity. This functionality minimizes the required customizations at deployment on various projects.

First, the main concepts are presented, including some fundamental ideas, arhitecture and functionalities. Some key proceedures are explained in detail, such as entity type analyzing and automatic editor generation. Finally, the configuration and deployment proceedure of the system is described.

Thesis concludes with description of some existing solutions and compares them with Parsek CMS 2.0. At the very end, achived results are evaluated and some ideas for future development are given.

Keywords:

Web application, Content Management System, CMS, Java EE, ORM, JPA, meta programming, GWT.

1. Uvod

Praktično vsa podjetja, katerih dejavnost je razvoj spletnih strani in spletnih aplikacij, morajo naročniku ponuditi tudi možnost urejanja vsebin oziroma poslovnih objektov. Za urejanje vsebin bolj predstavitveno naravnanih spletnih strani, kot so naprimer portali, je v zadnjem desetletju nastala cela množica sistemov CMS („*Content Management System*“), večina izmed njih pa poleg samega urejanja podatkov v bazi vključuje tudi mehanizme za prikaz vsebin oziroma izrisovanje spletnih strani s pomočjo takšnih ali drugačnih sistemov predlog („*templating*“). Tovrstni sistemi CMS tudi predvidevajo svoj vnaprej določen podatkovni model za hranjenje vsebin.

Med tem pa je za zahtevne poslovne aplikacije nemalokrat potrebno razviti lastno rešitev, ki je prilagojena funkcionalnostim in primerom uporabe te aplikacije. Naj kot primer izpostavim urejevalnik prodajnega artikla v spletni trgovini. Primerna zaslonska maska za to opravilo mora uporabniku ponujati hiter pregled (in možnost urejanja) najpomembnejših lastnosti, kot so glavna prikazna slika, naziv, opis, specifikacije in cena. Treba je omogočiti tudi povezovanje dodatnih storitev in izdelkov, ki bi jih kupci morda radi kupili zraven (t.i. „*upsell*“), in nenazadnje urejanje posebnih popustov za partnerje, itd... Srečamo se lahko torej z raznoraznimi poslovnimi objekti, katerih lastnosti bi želeli urejati. Logična posledica je želja po sistemu, ki bi to omogočal, brez da bi bile znova in znova potrebne nadgradnje, s katerimi bi se dodajalo podporo za posamezne vrste poslovnih objektov. Če povzamem, potrebujemo rešitev za upravljanje, ki se bo prilagajala podatkovnemu modelu ciljne aplikacije in ne obratno. Česa takšnega pa, začuda, do danes še nisem uspel najti, tako pravzaprav ni preostalo drugega, kot da se sam lotim razvoja.

Odločitev za razvoj lastnega sistema pa izvira tudi iz tradicije in izkušenj v podjetju. Naši prvi sistemi za upravljanje vsebin namreč segajo še na začetek stoletja, ko le-ti še niso bili prosto na voljo, sploh pa ne kot odprto-kodni projekti, kjer bi si izbrani sistem po potrebi lahko sami prilagodili in nadgradili.

Pri načrtovanju sem za podlago sicer še vedno vzel predhodni sistem Parsek CMC, vendar sem vključil sveže zamisli in poskusil odpraviti nekatere pomanjkljivosti, ki so bile tako izpostavljene oziroma zaželeno iz strani uporabnikov, kot tudi plod moje domišljije. Pri tem še omenimo, da gre v nasprotju s Parsek CMC, ki je del celovite platforme za razvoj spletnih aplikacij, vključno s tehnologijo za izris spletnih strani, pri Parsek CMS 2.0 zgolj za urejanje poslovnih objektov oziroma entitet in ne za celovito ogrodje za gradnjo spletnih strani oziroma aplikacij. Fokus je torej na urejanju poslovnih objektov in ne bogatih spletnih vsebin (čeprav je tudi slednje podprto, vendar ne tako učinkovito kot v namenskih sistemih CMS).

Pomembnejše zahteve, katerih zadovoljitev je bil cilj načrtovanja in razvoja:

- **Splošnost novega sistema** – želeli bi možnost urejanja katerihkoli entitetnih tipov,

brez da jih vnaprej poznamo in brez da za vsako ciljno spletno aplikacijo posebej prilagajamo CMS. Dobrodošla je tudi neodvisnost od SUPB.

- **Sobivanje s starejšimi tehnologijami** – predvsem zaradi migracije iz lastne CP2 platforme na sodobne odprto-kodne tehnologije se je pokazala potreba po novem sistemu CMS, ki bi bil sicer fokusiran na podporo novejših projektov, a hkrati obstaja možnost njegove uporabe na starih projektih.
- **Enostavna namestitev in konfiguracija** – same konfiguracije naj bo čim manj, potrebne parametre naj se poskuša samodejno nastaviti a vseeno obdržimo možnost, da jih pvozimo (v nekem smislu torej koncept „*convention over configuration*“). Če se da, se za konfiguracijo izognemo nepreglednim XML datotekam.
- **Spletni vmesnik in neodvisnost od platforme** – želimo si sodoben spletni vmesnik, kompatibilen z večino razširjenih brskalnikov ter nasploh uporabo prenosljivih tehnologij („*cross platform*“).
- **Moderne in poznane tehnologije** – uporabimo tehnologije, ki jih že poznamo in uporabljamo tudi pri razvoju novejših projektov, s čimer je zagotovljena še večja kompatibilnost s slednjimi, prav tako pa se ni potrebno naučiti veliko novega in se razvoj CMS-ja lahko takoj prične.

Glavni problemi so bili sledeči:

- **Kako razbrati podatkovni model** – samodejno zaznavanje entitetnih tipov, ki se jih bo urejalo, bi nas že pripeljala bliže splošnosti novega sistema, enostavne konfiguracije in možnosti sobivanja s starejšimi tehnologijami.
- **Kako opisati posamezne entitetne tipe** – entitetne tipe je treba na nek način opisati (katere in kakšne attribute imajo, kaj predstavljajo, itd...). Opisi morajo biti razumljivi tako strežniku kot odjemalcu oziroma morajo biti taki, da jih je mogoče obravnavati z izbranimi tehnologijami tako strežnika kot odjemalca.
- **Kako opisati instance entitet** – poskrbeti je treba tudi za vrednosti entitetnih atributov, ne samo za njihove opise. Ob začetku urejanja se morajo vrednosti prebrati iz podatkovne baze, nato pa prenesti od strežnika do odjemalca in po končanem urejanju spet nazaj ter se zapisati v podatkovno bazo.
- **Kako izmenjavati podatke med strežnikom in odjemalcem** – izbrati je treba način komunikacije med strežnikom in odjemalcem. Beseda „izbrati“ je nalašč uporabljena, ker je pri množici tehnologij, ki to omogočajo nesmiselno razvijati svoj protokol.
- **Zasnovati praktičen uporabniški vmesnik** – zamisliti si je potrebno neke osnovne funkcionalnosti (npr. ustvarjanje, brisanje, urejanje, iskanje, itd...) in koncepte (kot so naprimer zavihki, „*drag and drop*“, drevesna ureditev, itd...).
- **Kako zagotoviti urejevalnik, ki bo ustrezal katerikoli entiteti** – ker vnaprej ne vemo, kakšne entitete bodo urejane, bo reševanje tega problema kar velik izziv. Urejevalnik se bo očitno moral samodejno prilagajati atributom entitete.
- **Na kakšen način bo možna dodatna konfiguracija sistema** – kaj bo sploh treba konfigurirati in na kakšen način.

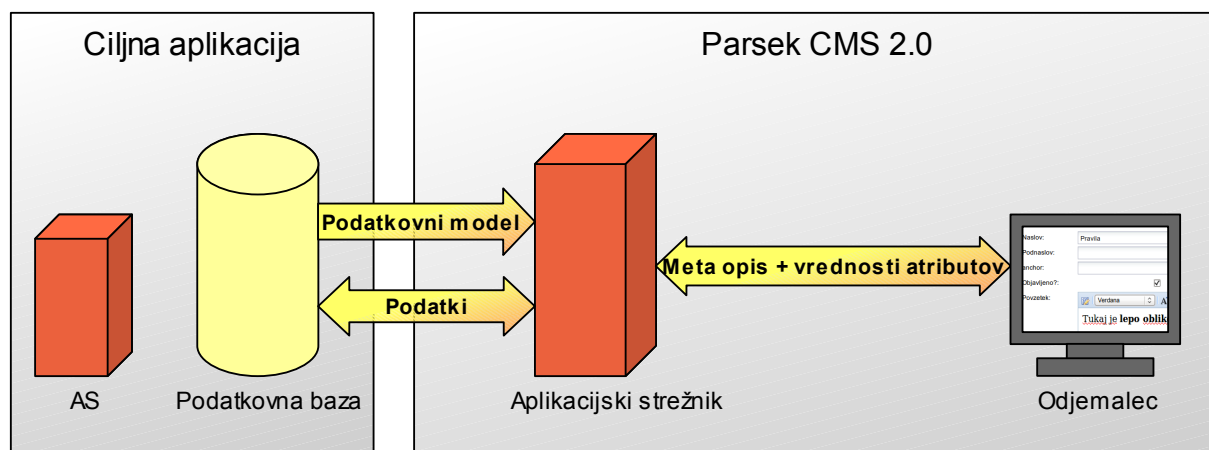
Naj na tem mestu še razjasnimo uporabo nekaterih pojmov v besedilu te naloge. Pojmi *tip vsebine* in *entitetni tip* pomenijo isto stvar. Podobno velja tudi za pojma *instanca vsebine* in *instanca entitete*. Tako enačenje obeh vrst poimenovanj si lahko privoščimo zato, ker je Parsek CMS 2.0 orodje, ki je namenjeno urejanju kakršnihkoli entitetnih tipov, tako da le-ti našemu sistemu pravzaprav predstavljajo vsebinske tipe.

2. Koncept rešitve

2.1 Arhitektura

Uporabljena je dokaj običajna tri-nivojska arhitektura, ki zajema podatkovno bazo, aplikacijski strežnik in odjemalca s spletnim uporabniškim vmesnikom, pri čemer pravzaprav Parsek CMS 2.0 zajema samo zgornja dva nivoja, saj uporablja podatkovno bazo ciljne aplikacije, katere poslovne objekte (oziroma entitete) bo urejal.

Zamisel je, da bi se aplikacija na strežniku povezala s ciljno podatkovno bazo in razbrala persistenčne entitetne tipe ter povezave med njimi. Ob tem bi sestavila nekakšne meta opise najdenih entitetnih tipov ter jih hranila v seznamu. Strežnik bi odjemalcu na zahtevo posredoval meta opis določenega entitetnega tipa, oziroma njegovih atributov, odjemalec pa bi na podlagi tega opisa lahko sproti zgeneriral urejevalnik z ustreznimi vnosnimi polji. Odjemalec mora seveda pri urejanju neke obstoječe entitete zgenerirana polja tudi zapolniti z že obstoječimi vrednostmi atributov, tako da mu strežnik poleg opisov atributov posreduje tudi njihove vrednosti. Ob končanju urejanja se komunikacija ponovi v obratni smeri in strežnik posodobi stanje entitete v podatkovni bazi.



Slika 1: Arhitektura in princip delovanja sistema

2.2 Drevesna kategorizacija vsebin

Drevesne organizacije in prikaza podatkov smo vajeni tako iz računalništva kot tudi iz

mnogih drugih področij. Z njim se vsakodnevno srečujemo in ga uporabljamo na tak ali drugačen način. Kot drevo so predstavljeni datotečni sistemi, kazala v knjigah s poglavji in podpoglavji, meniji in podmeniji v programih, zemljevidi spletnih strani („sitemap“) in podobno. Taka struktura nam je zato domača in jo lahko razumemo, ob pravilni uporabi pa je tudi pregledna in učinkovita.

Tudi vsebine spletnih strani in aplikacij smo v našem podjetju že od nekdaj uvrščali v nekakšno poljubno globoko drevesno strukturo kategorij in podkategorij. Podatkovni model, ki spremlja tako urejenost pa se do danes ni bistveno spremenil in je tako skupen skoraj vsem našim projektom. Tovrstna kategorizacija je zato podprta tudi v novem sistemu za urejanje vsebin, njen koncept pa si bomo sedaj natančneje pogledali.

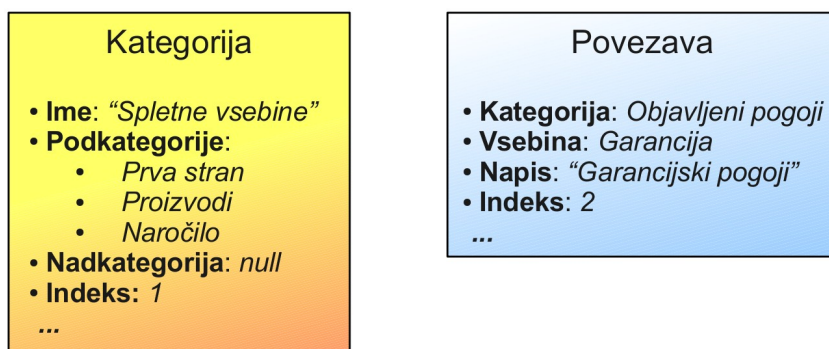
Kategorijo si lahko predstavljamo kot mapo (oziroma direktorij) v datotečnem sistemu, posamezno instanco vsebine pa kot datoteko. Tako kot mape lahko vsebujejo druge mape in raznorazne datoteke, lahko tudi kategorije vsebujejo (pod)kategorije in vsebine.

Zgornji opis je v resnici nekoliko poenostavljen. Vsebine niso dejansko vsebovane v kategorijah, temveč so z njimi samo povezane preko povezav. Tako je lahko ena instanca vsebine povezana v več različnih kategorij. Kategorije torej ne vsebujejo dobesedno instanc vsebin temveč samo povezave na njih (morda bi lahko spet uporabili primerjavo z datotečnim sistemom in rekli, da imamo v mapah zgolj simbolične povezave na dejanske datoteke).

Tak model naprimer omogoča, da se neka instanca vsebine prikazuje na več koncih naše spletne strani in ko jo je potrebno spremeniti, je potrebno le eno urejanje, pa bodo spremembe vidne povsod.

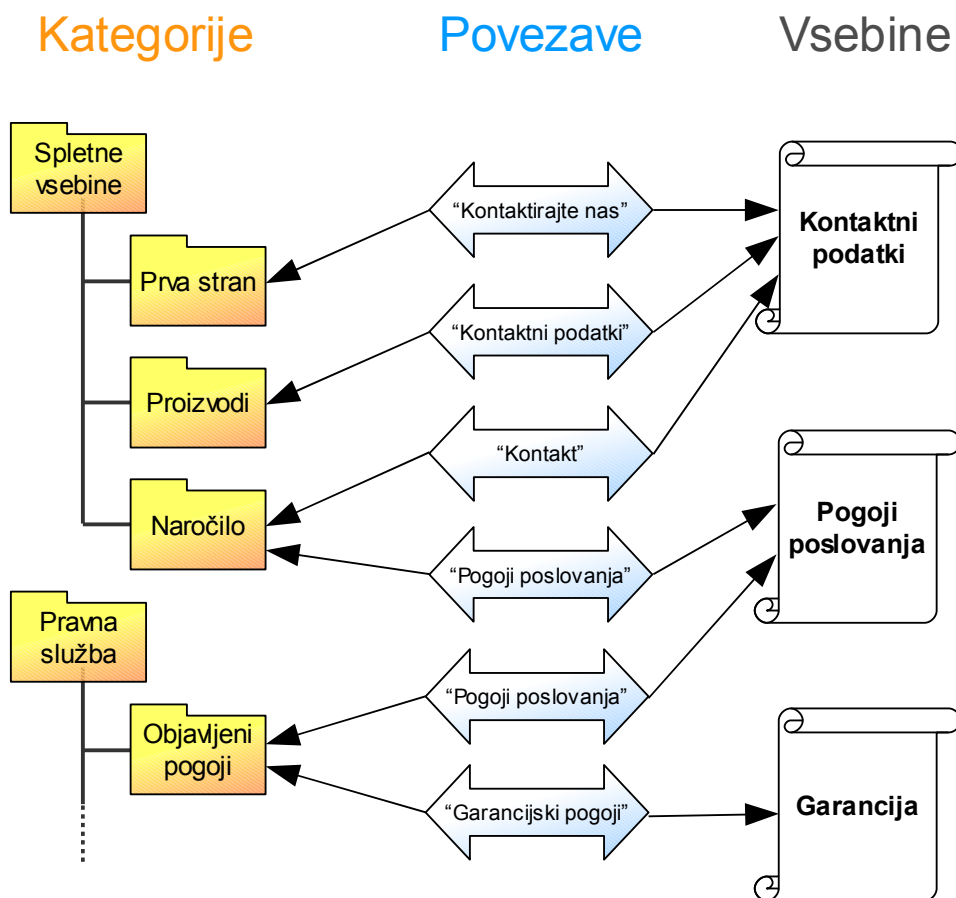
Poleg tega pa recimo tako lahko naredimo eno vejo kategorij, katerih vsebine so vidne na spletni strani, nato pa še nekaj vej kategorij, ki niso objavljene, ampak so namenjene zgolj preglednejši organizaciji oziroma hitrejši dostopnosti vsebin v samem sistemu za upravljanje.

Za lažje razumevanje je na naslednji strani diagram, ki ponazarja tukaj opisan koncept. Spodaj pa sta prikazana primera entitet iz tega diagrama, ki omogočata drevesno organizacijo.



Slika 2: Poenostavljen primer entitet, ki predstavljata kategorijo in povezavo

Indeks se lahko uporabi za vrstni red prikaza na spletni strani in v sistemu za urejanje, če je to pomembno. V diagramih so našteje samo osnovne lastnosti; v resnici jih je dosti več.



Slika 3: Primer drevesne kategorizacije vsebin

Hierarhija pod kategorijo Spletne vsebine je uporabljena za prikaz na spletni strani, med tem ko je kategorija Pravna služba namenjena zgolj za urejenost vsebin znotraj sistema za urejanje. Tako uporabniku sistema naprimer ni potrebno vedeti kje na spletni strani so objavljeni garancijski pogoji, ve pa da spadajo pod okrilje Pravne službe. Opazimo tudi, da je napis na posamezni povezavi lahko drugačen od naslova neke vsebine – povezave na Kontaktne podatke imajo tako naprimer vsakič drugačen napis. Seveda ni nujno, da se ta napis kjerkoli vidi – to je stvar implementacije ciljne aplikacije oziroma spletne strani.

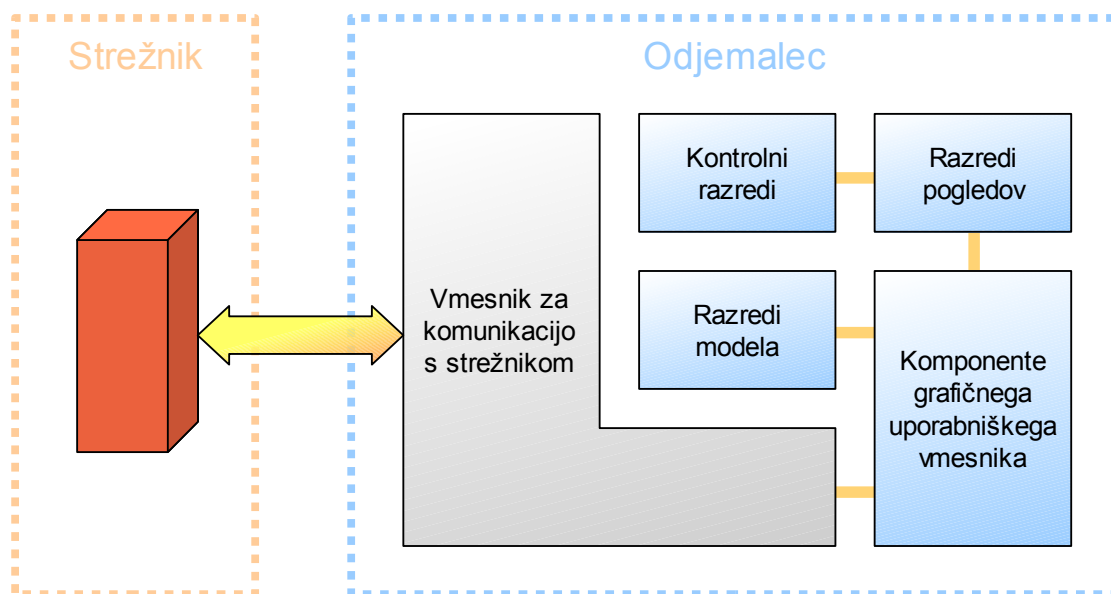
2.3 Odjemalec in uporabniški vmesnik

2.3.1 Arhitektura

Odjemalec je pravzaprav svoja aplikacija, ki „teče“ v spletnem brskalniku in ima določene gradnike, ki skupaj sestavljajo njeno arhitekturo, slednja pa še najbolj spominja na princip MVC („model-view-controller“), čeprav nekoliko odstopa od njega.

Kontrolne komponente sprejemajo in posredujejo razne dogodke drugim zainteresiranim

komponentam (običajno pogledom, včasih pa tudi drugim kontrolnim razredom). Dogodki vedno izvirajo iz nekega pogleda oziroma iz določene komponente uporabniškega vmesnika, saj jih povzročajo izključno uporabniki sami s svojimi akcijami. V nasprotju z običajnim MVC modelom, je večina poslovne logike odjemalca vsebovane kar v komponentah uporabniškega vmesnika in v kontrolnih komponentah, med tem ko so objekti modela namenjeni zgolj (začasnemu) hranjenju podatkov in samodejno ne posodablajo pogleda. Tudi razredi pogleda so precej „lahki“ saj je njihova naloga le povezovanje komponent grafičnega uporabniškega vmesnika v celovite zaslonske maske in posredovanje dogodkov.



Slika 4: Arhitektura odjemalca

2.3.2 Uporabniški vmesnik in funkcionalnosti

Drevo kategorij

Bralcu je verjetno poznan vsaj en pregledovalnik datotečnega sistema, naprimer Windows Explorer, ki na levi prikazuje drevesno strukturo map, na desni pa seznam v trenutno izbrani mapi vsebovanih datotek. Podoben pristop je uporabljen tudi za Parsek CMS 2.0. Na levi je torej prikazano drevo kategorij, na desni pa seznam povezav, ki z izbrano kategorijo povezujejo instance vsebin (na Sliki 5 je za lažjo predstavo skica uporabniškega vmesnika).

Zavihki

Moderni brskalniki so popularizirali zavihke („tabs“), ki so se izkazali za zelo praktičen del uporabniških vmesnikov. Tako kot brskalniki z zavihki omogočajo, da imamo hkrati odprtih več spletnih strani, Parsek CMS 2.0 omogoča, da imamo hkrati za urejanje odprtih več entitet. Zavihki se odpirajo desno od drevesa kategorij, prvega zavihka pa se ne da zapreti in

prikazuje prej omenjen seznam povezav iz izbrane kategorije. Ob dvokliku na eno od teh povezav, se odpre nov zavihek z urejevalnikom za ciljno vsebino.

Napredno iskanje

Drevo kategorij lahko skrijemo in namesto njega prikažemo vnosno masko za napredno iskanje entitet. Uporabnik izbere po katerem entitetnem tipu bo iskal, nato pa še kateri atributi bodo vključeni v iskanje in katere vrednosti morajo zavzemati. Na desni so potem namesto zavihkov prikazani rezultati iskanja, ob dvokliku na enega izmed rezultatov pa se vrnemo na drevo kategorij, na desni pa se ponovno prikažejo zavihki, vključno z novim zavihkom, v katerem je za urejanje odprt izbran rezultat iskanja.

Napredno iskanje **išče tudi po entitetah, ki niso povezane v nobeno kategorijo**, kar omogoča urejanje podatkov tudi za aplikacije, ki slednjih nimajo kategoriziranih. S tem Parsek CMS 2.0 postane še bolj splošno orodje za urejanje vsebin, saj bi ga lahko uporabili tudi na projektih, ki odstopajo od standardnega vzorca kategorij. Vendar pa je scenarij uporabe na takih projektih zelo malo verjeten, ker jih v našem podjetju skorajda ni, tistih nekaj posebnitev pa ima že lastne sisteme za urejanje vsebin.

Urejevalniki

Urejevalnik je, kot je bilo že večkrat omenjeno, dinamično zgeneriran, ko naložimo neko vsebino za urejanje. Postopek generiranja bi na kratko lahko opisali kot iteriranje čez meta opise atributov entitete, pri čemer za vsakega zgeneriramo ustrezno vnosno polje in ga dodamo na zaslonsko masko. Nazadnje pa zgenerirana polja še zapolnimo z vrednostmi. Če gre za novo ustvarjeno entiteto polja seveda ostanejo prazna.

Poleg splošnega urejevalnika pa so v sistem vključeni še nekateri prilagojeni urejevalniki, ki olajšajo urejanje entitet nekaterih vnaprej predvidenih entitetnih tipov, naprimer urejevalnik za slike ali pa urejevalnik za prodajne artikle v spletni trgovini. S slednjimi entitetnimi tipi bi sicer brez problemov lahko opravi tudi splošni urejevalnik – prilagojeni so bili dodani zaradi boljše uporabniške izkušnje in ne zaradi kakšnih tehničnih ovir.

Splošno iskanje

Poleg naprednega iskanja je na voljo tudi hitro dostopno vnosno polje, kamor vnesemo iskalno poizvedbo in sprožimo iskanje s tipko *Enter*. Rezultati splošnega iskanja se odprejo kar v novem zavihku, posamezen rezultat pa lahko tudi tukaj z dvoklikom odpremo za urejanje. Pri splošnem iskanju gre za t.i. „*full-text-search*“ po indeksiranih vsebinah. V to iskanje so hkrati vključene vsebine različnih tipov. Kateri entitetni tipi in kateri atributi teh entitetnih tipov so vključeni v indeks ter posledično v splošno iskanje, je potrebno posebej skonfigurirati.

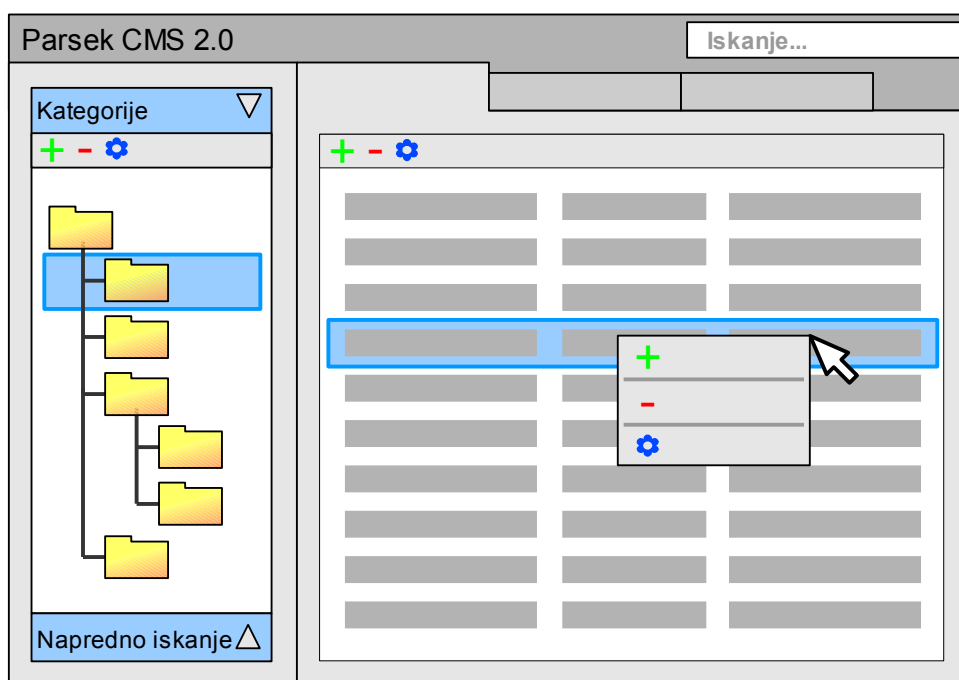
Povleci in spusti (drag and drop)

Nekatere operacije je lažje in bolj intuitivno izpeljati z uporabo vsem dobro znanega koncepta „povleci in spusti“. Tako recimo lahko preurejamo drevo kategorij, naprimer spremenimo

vrstni red podkategorij znotraj neke kategorije (spremenimo na Sliki 2 omenjen atribut Indeks), ali pa celotno poddrevo premaknemo v neko drugo kategorijo. Preurejamo lahko tudi vrstni red povezav v neki kategoriji, itd...

Orodne vrstice in kontekstualni meniji

Operacije s trenutno označenimi elementi uporabniškega vmesnika lahko izvajamo z izbiro ustreznih gumbov v orodnih vrsticah ali s klikom na desni miškin gumb, s čimer se odpre ustrezen kontekstualni meni. Tudi s tema dvema koncepta se srečujemo na različnih mestih uporabniškega vmesnika, kjer je pač smiselno in intuitivno.



Slika 5: Skica uporabniškega vmesnika

Na levi drevo kategorij, namesto katerega lahko po potrebi razpremo Napredno iskanje. Na desni zavihki s trenutno fokusiranim prvim zavihkom, ki prikazuje seznam povezav (na instance vsebin), ki jih vsebuje trenutno izbrana kategorija. Desni klik na eno od povezav odpre kontekstni meni. Na vrhu drevesa kategorij in seznamu povezav sta orodni vrstici. Zgoraj desno je vnosno polje za splošno iskanje.

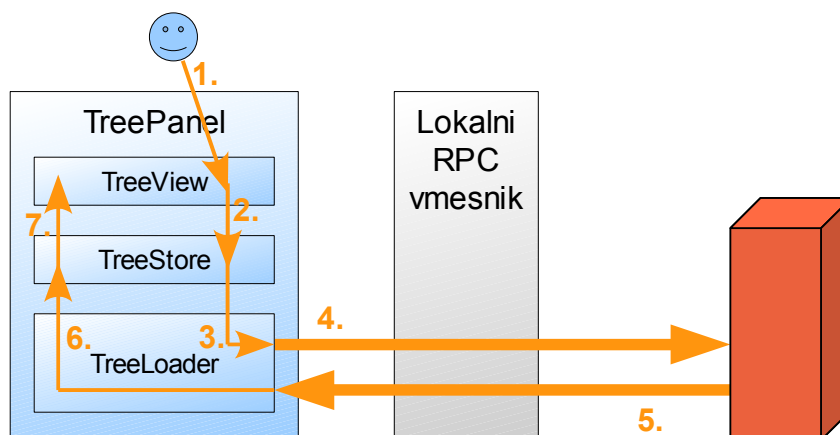
2.3.3 Izbira tehnologije

Odjemalec je napisan z ogrodjem GWT (Google Web Toolkit) in dodatkom Ext GWT (na kratko GXT). Prvi omogoča programiranje odjemalca vključno z uporabniškim vmesnikom

kar v programskem jeziku Java, GWT prevajalnik pa kodo nato pretvori v kombinacijo HTML-ja, CSS-jev in JavaScript-a, ki je kot celota kompatibilna z večino modernih brskalnikov. Ta funkcionalnost eliminira potrebo po oblikovanju in razrezu, razvijalec pa se lahko osredotoči na samo funkcionalnost in ne na podrobnosti v zvezi z izgledom ter razlikami med brskalniki. Poleg tega nam omogoča uporabo objektno usmerjenega programiranja in naprednih programskih konstruktov, kar bi s samim JavaScript-om bilo bolj nepregledno in manj enostavno. GXT pa je dodatek za GWT, ki vključuje veliko GUI elementov in mehanizmov za še hitrejši razvoj uporabniškega vmesnika. Še ena od pomembnih funkcionalnosti GWT-ja je podpora za oddaljene klice (RPC - „remote procedure call“), s čimer je rešen problem komunikacije odjemalca s strežnikom.

Uporabo kombinacije GWT+GXT za administrativna orodja na kompleksnejših projektih sicer narekujejo tudi interni standardi oziroma predpisi v podjetju glede izbire razvojnih tehnologij.

Na Sliki 4, ki prikazuje arhitekturo odjemalca, se morda zdi nekoliko nenavadna povezava med komponentami grafičnega uporabniškega vmesnika ter vmesnikom za komunikacijo s strežnikom in si zato zasluži krajše pojasnilo. GXT grafične komponente (*Widgets*), kot so na primer preglednice (uporabljene pri seznamu povezav, vsebovanih v neki kategoriji), padajoči meniji, drevesa (drevo je uporabljeno za prikaz drevesa kategorij) in druge, so sestavljene iz treh delov – pogleda, shrambe podatkov in nalagalnika podatkov. Slednji je opcijski, saj lahko vsebino shrambe podatkov, ki jih prikazuje pogled, spreminjamo tudi sami programsko. Naslednji diagram prikazuje, kako funkcionira komponenta *TreePanel*, ki se uporablja za prikazovanje dreves.



Slika 6: Brskanje po drevesu

Uporabnik razpre kategorijo (1). Pogled poskuša iz lokalnega podatkovnega skladišča dobiti podatke o podkategorijah (2). Če jih skladišče še nima, jih slednje zahteva od nalagalnika (3), ki naredi RPC zahtevek na strežnik (na TreeService) (4). Po prejemu odgovora (5) se podkategorije shranijo v lokalnem skladišču (6) in se posodobi pogled (7).

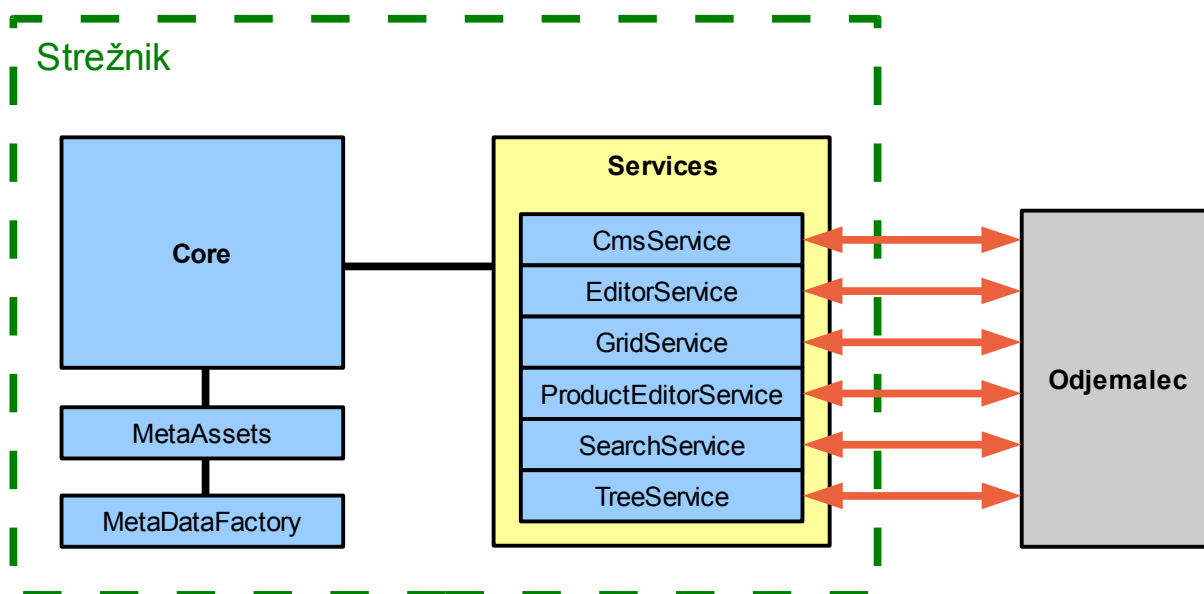
Med grafične komponente uporabniškega vmesnika pa spadajo tudi tiste, ki sem jih sam razvil in združujejo razne že obstoječe komponente GXT-ja ali pa sem obstoječe komponente

razširil z novimi funkcionalnostmi. Tudi takšne komponente komunicirajo s strežnikom in nanj pošiljajo raznorazne zahteve.

2.4 Strežnik

Strežniška aplikacija ima v splošnem dve nalogi. Prva je pregledati podatkovni model v ciljni podatkovni bazi in iz njega **sestaviti meta opise** entitetnih tipov ter njihovih atributov. Druga pa je **izpolnjevanje zahtev odjemalca**, ki vključuje tudi pretvorbe entitet v obliko, primerno za posredovanje in obdelavo v odjemalcu ter obratno. Izpolnjevanju zahtev odjemalca so namenjeni t.i. servisi, ki implementirajo metode, katere so odjemalcu na voljo preko klicev RPC. Poleg samega izpostavljanja teh metod, je v njih implementiran tudi dovršen del programske logike, ki izpolnjuje odjemalčeve zahteve, med tem ko so jedru (*Core*) delegirane bolj osnovne operacije. Komponenta *EditorService*, ki s svojimi oddaljenimi metodami nudi podporo urejevalniku, tako naprimer uporablja *Core* za nalaganje in shranjevanje entitet, med tem ko spremembe in raznorazne obdelave na njih izvaja sama. Servisi torej implementirajo logiko, s katero podpirajo določene komponente ali primere uporabe v odjemalcu, med tem ko so v jedru implementirane osnovnejše funkcije, ki se jih servisi poslužujejo.

MetaAssets hrani seznam meta opisov entitetnih tipov, ki so na voljo v ciljnem podatkovnem modelu. Na zahtevo jih sestavi komponenta *MetaDataFactory*, ki implementira logiko za prebiranje ciljnega podatkovnega modela in sestavljanje meta opisov. Servisi lahko do meta opisov dostopajo preko jedra, ki upravlja s komponento *MetaAssets*.



Slika 7: Glavne komponente strežnika

Pomembnejše metode razreda *Core* so naštet in opisane v *Prilogi D*, posamezne storitve (services) pa so podrobneje predstavljene v *Prilogi F*.

2.4.1 Izbira tehnologij

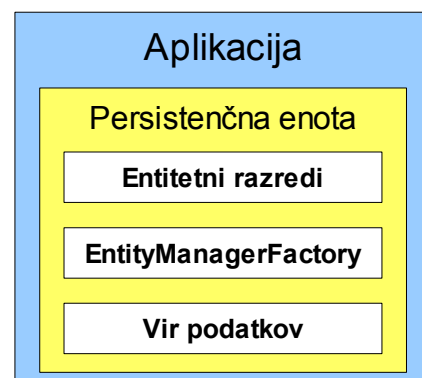
Nekateri dejavniki, ki so vplivali na izbor tehnologij izvirajo kar iz samih zahtev, podanih v uvodu. Poskušamo torej izbrati tehnologije in ogrodja, ki bodo v čim večji meri že same po sebi pokrivala našete zahteve. Tako naprimer za prebiranje podatkovnega modela lahko uporabimo eno izmed številnih ogrodij za ORM („*object-relation mapping*“), ki tovrstno funkcionalnost v večji meri že podpirajo, itd... Enako velja za zahteve po prenosljivosti, sobivanju s tehnologijami obstoječih projektov in podobno.

Aplikacije, katerim je novi CMS v prvi vrsti namenjen, so (oziroma bodo) napisane kot Javanske spletne aplikacije. Java je prenosljiva, konfigurabilna, zrela in ima razvito podporo za meta programiranje. V podjetju se večinoma uporablja aplikacijski strežnik JBoss AS in Java EE tehnologija v povezavi z ogrodjem Seam. Seam „pod eno streho“ združuje tehnologije Java EE, JSF, JPA, Hibernate, EJB3 in druge. Poleg tega pa implementira še svoje mehanizme za hiter razvoj spletnih aplikacij (komponentni model, „*dependency injection*“, varnostni mehanizmi, itd...).

Za neodvisnost od SUPB skrbi ORM vmesnik JPA („*Java Persistence API*“), bolj natančno ogrodje Hibernate, ki je ena od implementacij JPA vmesnika. Namesto s tabelami in SQL poizvedbami raje delamo z objekti (entitetami) in poizvedbenim jezikom EJB-QL (jezik, podoben SQL, vendar je namenjen delu z objekti), Hibernate pa samodjelo generira in izvaja ustrezne poizvedbe na podatkovni bazi. Entitetni razredi so običajni Javini razredi (POJO – „*plain old java object*“) z lastnostmi, ki predstavljajo entitetne attribute. Z anotacijami na entitetnem razredu in njegovih lastnostih podamo definicijo posamezne entitete (npr. povemo kateri atribut je primarni ključ, podamo povezave na druge tabele, bolj natančno določimo tipe atributov – v osnovi so sicer slednji določeni že s samim tipom lastnosti, itd...). Anotacije so mnogo bolj praktične in pregledne kot obsežni XML-ji. Primer preprostega entitetnega razreda je za lažjo predstavo s komentarji podan v *Prilogi A* in bralcu priporočam, da si ga vsaj na hitro pogleda.

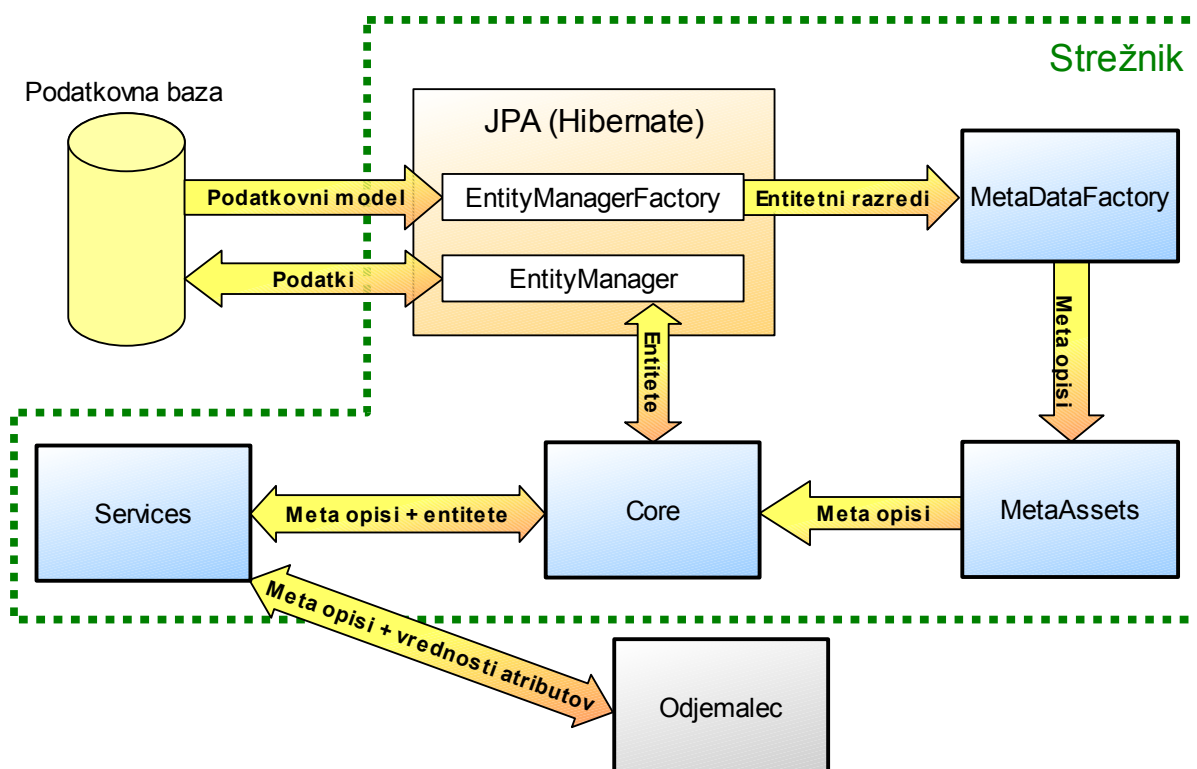
Do entitetnih razredov lahko pridemo na 2 načina. Lahko jih sami napišemo, saj običajno med načrtovanjem izdelamo tudi konceptualni podatkovni model, tako da je pretvorba v kodo večinoma enostavna in je s strani nekaterih CASE orodij tudi avtomatizirana. Hibernate nato ob namestitvi oziroma zagonu aplikacije lahko samodejno posodobi ali kreira podatkovno shemo v bazi, da ustreza objektnemu modelu. Tak pristop je zelo pogost pri novih projektih. Druga možnost pa je uporaba orodja „hbm2java“, ki deluje v obratni smeri in sicer iz obstoječe sheme v podatkovni bazi zgenerira entitetne razrede, ki jih nato lahko uporabljamo v aplikaciji.

Z entitetami upravlja poseben objekt razreda *EntityManager*. Le-ta upravlja s stanji vseh „managed“ entitet, jih sinhronizira s podatkovno bazo, izvaja raznovrstne poizvedbe in je navidez (iz vidika programiranja) glavna vez med aplikacijo in persistenčnim nivojem. Za instanciranje *EntityManager*-ja skrbi komponenta *EntityManagerFactory*. Slednja je del neke persistenčne enote (*persistence-unit*), le-ta pa poleg *EntityManagerFactory*-ja vsebuje še vir podatkov, ki je



večinoma podatkovna baza (na račun abstrakcije JPA bi bilo teoretično lahko tudi kaj drugega) in entitetne razrede, ki se nahajajo znotraj aplikacije, katere del je persistenčna enota. Ena aplikacija sicer lahko uporablja tudi več persistenčnih enot, vendar to ni pogosto. CMS uporablja trik, s katerim so mu na voljo entitete ciljne aplikacije. Pred namestitvijo namreč nastavimo konfiguracijo njegove persistenčne enote tako, da uporablja *EntityManagerFactory* ciljne aplikacije, kar je v resnici stvar ene spremenljivke v konfiguraciji, s katero je določeno ime *EntityManagerFactory*-ja, tako da je treba v konfiguraciji le navesti enako ime in aplikacije si bodo delile isto instanco slednjega. Tako so entitete ciljne aplikacije na voljo tudi sistemu CMS. Podrobnosti v zvezi s tem si bomo pogledali nekoliko kasneje, vključno z rešitvijo za tiste ciljne aplikacije, ki ne uporabljajo JPA ali pa za namestitev CMS-ja na ločen strežnik. V teh dveh primerih se prej omenjenega trika seveda ni možno posluževati.

Hibernate ob zagonu aplikacije pregleda vse razrede iz aplikacije in ugotovi kateri so entitetni. Seznam vseh entitetnih razredov iz persistenčne enote je nato na voljo skozi metode razreda *EntityManagerFactory*. Ko imamo enkrat entitete v obliki Javinih razredov, pa lahko z „Java Reflection API“ (Javino ogrodje za meta programiranje) pregledujemo njihovo strukturo (naprimer tipe lastnosti, anotacije na lastnostih, itd...) in tako razberemo podatkovni model ter sestavimo meta opise, ki bodo primerni za pošiljanje odjemalcu in obdelavo na odjemalčevi strani.



Slika 8: Povezanost komponent

2.5 Meta opisi in predstavitev vrednosti atributov

Ko sistem dela z entitetami, dela z dvema vrstama podatkov, povezanih z njimi. Ena so opisi entitetnih tipov oziroma njihovih atributov, drugo pa so dejanske vrednosti entitetnih atributov. Entitetni tipi so opisani z instancami razreda *MetaAsset*, njihovi atributi pa z instancami razreda *MetaField*. Vrednosti atributov neke entitete pa se shranijo v objekt razreda *Asset*.

V spodnji tabeli je podano, kako so opisi podatkov in podatki sami predstavljeni v različnih kontekstih oziroma okoljih.

Logični model relacijske P.B.	Konceptualni model entiteta-razmerje	Parsek CMS 2.0	
		JPA, Strežnik	Strežnik, Odjemalec
Shema relacije	Entitetni tip	Definicija entitetnega razreda z anotacijami	MetaAsset in MetaField
N-terica relacije	Entiteta	Instanca (objekt) entitetnega razreda	Asset

Morda se bralec sedaj že sprašuje, zakaj je pravzaprav sploh potrebno pretvarjanje v meta opise in *Asset*, oziroma zakaj se raje ne uporabi kar povsod JPA entitetnih razredov, saj je bilo vendar rečeno, da je tudi odjemalec napisan v Javi. Razlog tiči v tehnologiji GWT, ki v resnici ne implementira celotnega nabora razredov oziroma funkcionalnosti, ki so sicer na voljo v Java SE, temveč samo neko podmnožico. Med drugim, kar je pravzaprav najpomembnejše, manjka tudi Java Reflection API, s katerim bi lahko odjemalec sam analiziral entitetne razrede in generiral ustrezne urejevalnike. Zaradi pomanjkanja te funkcionalnosti, je potrebno to storiti že na strežniku in rezultate analize v primerni obliki (torej kot meta opise) posredovati odjemalcu.

V naslednjih treh razdelkih so opisane tiste lastnosti razredov *MetaAsset*, *MetaField* in *Asset*, ki so pomembne za razumevanje koncepta. Podrobnejši opisi z bolj popolnim seznamom lastnosti teh razredov pa je na voljo v *Prilogi B*.

2.5.1 MetaAsset

V njem so podani osnovni podatki o opisanem entitetnem razredu in seznam z opisi atributov.

Lastnost	Pomen
Ime	Ime opisanega entitetnega tipa, ki je kar enako imenu entitetnega razreda.
Seznam atributov (seznam <i>MetaField</i> -ov)	Vsak vnos v seznamu opisuje enega od atributov in hkrati določa kakšno vnosno polje bo zanj zgenerirano v urejevalniku.
Atribut za primarni ključ	Pove, kateri od atributov iz seznama predstavlja primarni ključ.

Atribut za naslov (oziroma napis)	Pove, kateri od atributov iz seznama naj se uporablja kot naslov oziroma napis, ki bo v uporabniškem vmesniku prikazan na raznih seznamih, itd... Kako se ta atribut določi oziroma izbere, je razloženo kasneje.
--	---

2.5.2 MetaField

Opisuje določen atribut nekega entitetnega tipa in hkrati tudi vnosno polje, ki bo v urejevalniku zgenerirano in uporabljano za urejanje vrednosti tega atributa.

Lastnost	Pomen
Ime	Ime tega atributa, ki je enako imenu ustrezne lastnosti iz entitenega razreda.
Tip atributa (oziroma vnosnega polja)	Pove kakšnega tipa je atribut in implicitno tudi kakšno vnosno polje potrebujemo. Ali gre na primer za številsko vrednost, znakovni niz, datum, morda pa kar za nek drug entitetni razred ali celo seznam entitet, itd...
Tip vnosnega polja	V nekaterih primerih informacija o tipu atributa ni dovolj, da bi se algoritem lahko odločil, katero vnosno polje je potrebno. Takrat se opre še na to polje.
Obvezna vrednost	Ali mora biti vrednost obvezno prisotna in zato vnosno polje ne sme ostati prazno.
Podtip	Kadar opisani atribut predstavlja povezavo z neko drugo entiteto (ali celo več njih, na primer pri <i>one-to-many</i> razmerjih), ta lastnost določa katerega tipa je ciljna entiteta (oziroma so ciljne entitete).

2.5.3 Asset

Če sta prejšnja razreda namenjena shranjevanju opisa entitetnega tipa oziroma entitetnega razreda, je ta razred namenjen shranjevanju dejanskih vrednosti posameznih atributov, torej stanja neke entitete.

Lastnost	Pomen
Primarni ključ	Vsebuje vrednost primarnega ključa.
Tip	Vsebuje ime ustrežujočega <i>MetaAsset</i> -a.

Naslov (oziroma napis)	Vrednost atributa, ki je uporabljen za naslov.
Seznam atributov (seznam <i>MetaField</i> -ov)	Gre za kopijo tovrstnega seznama iz pripadajočega <i>MetaAsset</i> -a.
Vrednosti atributov	Razpršena tabela, ki vsebuje vrednosti atributov, kot ključe pa uporablja imena atributov.

Opazimo, da *Asset* ne vsebuje celotnega opisa entitetnega tipa ampak le seznam z opisi polj. Opis samega entitetnega tipa (*MetaAsset*) namreč vsebuje še nekatere lastnosti, ki jih uporablja predvsem strežnik pri nekaterih operacijah in tako ni potrebe, da bi jih vedno prenašali med njim in odjemalcem. Posledično in zaradi priročnosti v *Asset*-u hranimo tudi vrednost primarnega ključa in naslova, sicer bi morali najprej iz *MetaAsset*-a dobiti informacijo, katera atributa vsebujeta omenjeni vrednosti in jih nato prebrati iz razpršene tabele.

Naj še slednji namenimo nekaj besed. Gre za običajno Javino implementacijo razpršene tabele, definirano z *HashMap<String, Object>*. Za ključ je uporabljeno ime atributa, pripadajoča vrednost pa je kar vrednost tega atributa.

2.5.4 Primer

Naj do sedaj povedano v tem podpoglavju povzamem še s preprostim primerom, ki predpostavlja, da v podatkovni bazi hranimo podatke o nekih uporabnikih in sicer njihov elektronski naslov, geslo in letnico rojstva.

1.) Podatkovna baza:

Ime tabele: Uporabnik		
Email <i>varchar(255) (PK)</i>	Geslo <i>varchar(20) (not null)</i>	Letnik <i>int</i>
„x@y.z“	„1234“	1985

2.) JPA entitetni razred in objekt:

```
@Entity
@Table(name = "Uporabnik")
public class Uporabnik {

    @Id @Column(name = "Email", length = 255)
    String email;

    @Column(name = "Geslo", length = 20, nullable = false)
    String geslo;

    @Column(name = "Letnik")
    int letnik;

    //getters and setters for properties here
}
```

Uporabnik: xyz
email: „x@y.z“
geslo: „1234“
letnik: 1985

3.) Parsek CMS 2.0 meta opisi in vrednosti atributov:

MetaAsset: maUporabnik Ime: „Uporabnik“ Seznam atributov: {mfEmail, mfGeslo, mfLetnik} Atribut za primarni ključ: email Atribut za naslov: email (meta opis entitetnega tipa Uporabnik)	MetaField: mfEmail Ime: „email“ Tip atributa: FieldType.STRING Tip vnosnega polja: InputType.GUESS Obezna vrednost: true Podtip: null (meta opis atributa email)	MetaField: mfGeslo ... MetaField: mfLetnik ...
Asset: aUporabnik Primarni ključ: „x@y.z“ Tip: „Uporabnik“ Naslov: „x@y.z“ Seznam atributov: {mfEmail, mfGeslo, mfLetnik} Vrednosti atributov: { „email“→„x@y.z“, „geslo“→„1234“, „letnik“→1985 } (vrednosti atributov entitete Uporabnik)		

2.6 Standardni tipi vsebin

Skoraj ni projekta, kjer ne bi bilo treba urejati standardnih spletnih vsebin, kot so članki, v katerih so seveda tudi slike in povezave na raznorazne datoteke (najpogosteje na kakšne PDF dokumenti in podobno). S časom so se tudi entitetni tipi, ki tovrstne vsebine hranijo, stabilizirali oziroma prenehali spreminjati in dopolnjevati ter jih v isti obliki uporabljamo na vseh projektih. Zdi se smiselno, da so kot taki že privzeto vključeni v CMS sistem in je njihovo urejanje podprto. V podatkovni bazi se za te entitetne tipe ob namestitvi CMS-ja samodejna kreira ustrezna shema, če še ne obstaja.

Vsi osnovni tipi med drugim vsebujejo attribute ki hranijo datum nastanka, datum zadnje spremembe, datum objave, uporabnika, ki je vsebino ustvaril in uporabnika, ki jo je nazadnje spremenil.

Članek („Article“)

Članek ponavadi predstavlja vsebino točno določene strani oziroma zaslonske maske s statično vsebino ali posamezen del vsebine, iz katere je sestavljena. Po potrebi ga lahko seveda uporabimo kot marsikaj drugega, saj je stvar posamezne aplikacije katere attribute bo izkoristila in na kakšen način. Nemalokrat se (morda nekoliko potratno) uporabi samo eden ali nekaj atributov. Primeri atributov članka so naslov, povzetek, vsebina, povezane slike, povezani članki, vir, itd...

Datoteka („Binary“)

Predstavlja datoteko, ki jo naložimo preko sistema CMS. Običajno se take datoteke nalagajo s

tem namenom, da kasneje do njih vodi neka povezava, naprimer iz članka. Še en pogost način uporabe so FLV (*flash video*) datoteke, ki jih nekje predvaja v spletno stran vgrajen predvajalnik. Pomembnejši atributi so naslov datoteke (naprimer „Obrazec za pristopno izjavo“ v primeru PDF dokumenta), velikost datoteke, ime datoteke, tip vsebine datoteke (MIME tip), vrednost zgoščevalne funkcije in sama vsebina datoteke.

Slika („Image“)

Je pravzaprav razširitev prej opisanega tipa „Binary“. Vsebuje še dodatne attribute, ki opisujejo za sliko specifične lastnosti, kot naprimer dimenzije in tekstovni opis slike.

Konfiguracija („Configuration“)

Vsebuje ime konfiguracije in pare ključ-vrednost. Tako ključi kot vrednosti so znakovni nizi. Kot primer uporabe si lahko zamislimo teme za izgled spletnega mesta, kjer bi si uporabnik lahko izbral zelen, moder ali rdeč izgled, v bazi podatkov pa bi imeli temu ustrezno shranjene različne konfiguracije z imeni „Zelena“, „Modra“ in „Rdeča“. Ključi bi pomenil nek vizualen element na spletni strani, vrednosti pa njegovo barvo, sliko ozadja, itd...

Uporabnik („User“)

Hrani podatke o uporabnikih, ki se lahko prijavijo in uporabljajo CMS. Entiteta je sicer povzeta iz prejšnjih sistemov za urejanje, vendar se v novem še ne uporablja v celoti. Manjka naprimer podpora sistemu pravic, ipd... Take funkcionalnosti so predvidene za prihodnje verzije. Se pa naprimer na posamezni vsebini zabeleži kdo jo je ustvaril in kdo nazadnje uredil.

Izdelek („Tangible“)

Ta tip sam po sebi ni vključen v distribucijo CMS-ja, vendar se uporablja v naših spletnih trgovinah in predstavlja prodajni artikel, zato je v sistemu vseeno posebej obravnavani. Za entitetni tip „Tangible“ obstaja v CMS-ju poseben urejevalnik, ki naredi urejanje prodajnih artiklov hitreje in preglednejše. Iz teh razlogov ga na tem mestu vseeno omenjam. Ima malo morje atributov, zato jih tukaj ne bi našteval.

3. Realizacija in delovanje

V prejšnjem poglavju so bili podani osnovni koncepti in naštetih nekateri pomembni procesi, v tem poglavju pa so slednji konkretnije opisani. Največ pozornosti bo posvečeno analizi entitetnih razredov, pretvarjanju entitet v *Asset*-e in dinamičnemu generiranju urejevalnika. Omenjeni procesi pa niso samo pomembni ampak tudi nekoliko zapleteni, njihova implementacija pa obsežna, zato bo ob njihovi razlagi bralec pogosto usmerjen še v priloge, kjer so razložene podrobnosti implementacij, naprimer v obliki izvorne kode s komentarji, raznih preglednic, itd...

3.1 Inicializacija uporabniške seje

Uporabnika na začetku pričaka preprosta vnosna maska, kamor vpiše uporabniško ime in geslo ter se tako prijavi v sistem. Ob navidez preprostem procesu, pa se na strežniku po uspešni avtentifikaciji poleg začetka nove uporabniške seje zgodi še nekaj važnih operacij.

Kreira se **nova instanca jedra** (*Core*), ki nato poskrbi za inicializacijo preostalih komponent, med drugim tudi komponente *MetaAssets*. Slednja ob inicializaciji uporabi *MetaDataFactory*, ki analizira entitetne razrede in **ustvari meta opise entitetnih razredov** ter jih shrani v seznam, ki je nato na voljo jedru in posredno preko jedra še servisom.

Vsak uporabnik ob prijavi torej dobi svojo instanco jedra s pripadajočimi komponentami, ki je asociirano z uporabniško sejo. Jedro hrani tudi določeno stanje v zvezi z uporabnikovo sejo in s tem, kaj trenutno počne, zato tudi ni skupno vsem uporabnikom.

3.1.1 Analiza entitetnih razredov

Analiza entitetnih razredov se prične v konstruktorju razreda *MetaAssets*, ki v psevdo kodi izgleda tako:

```
public MetaAssets(EntityManagerFactory em)

    seznamEntitetnihRazredov = em.vrniVseEntitetneRazrede();

    Za vsak entitetni razred {
        MetaAsset ma = MetaDataFactory.createMetaAsset(entitetni razred);
        seznamMetaOpisov.dodaj(ma);
    }

}
```

Vsak najden entitetni razred se torej posreduje ustrezni metodi razreda *MetaDataFactory*, ki

ga analizira in ustvari ter vrne njegov opis *MetaAsset*. Poenostavljena psevdo koda te metode je sledeča:

```
MetaAsset createMetaAsset(entitetni razred) {

    MetaAsset ma = new MetaAsset();

    Nastavi ime MetaAsset-a ter nekatere druge lastnosti.

    seznamLastnostiRazreda = uporabi Java Reflection API za pridobitev
        seznama lastnosti razreda.

    Za vsako lastnost razreda {

        Če je lastnost tranzitivna (ni persistenčna), nadaljuj z naslednjo
        iteracijo zanke!

        MetaField mf = createMetaField(lastnost razreda);

        ma.addField(mf);

    }

    Sedaj, ko poznamo vse persistenčne attribute, nastavi še:
        1.) Opis atributa, ki predstavlja naslov.
        2.) Opis atributa, ki predstavlja primarni ključ

    return ma;

}
```

Najprej torej poskrbimo za tiste lastnosti meta opisa, ki jih lahko direktno razberemo iz entitetnega razreda, nato iteriramo čez vse lastnosti entitetnega razreda in uporabimo posebno metodo, ki nam sestavi opis persistenčne lastnosti oziroma entitetnega atributa ter ta opis dodamo na seznam. Nazadnje, po izteku zanke, nastavimo še tiste lastnosti meta opisa, za katere moramo najprej poznati vse attribute.

Psevdo koda metode, ki sestavi opis atributa pa izgleda takole:

```
MetaField createMetaField(lastnost entitetnega razreda) {

    returnType = uporabi Java Reflection API in ugotovi katerega Javinega
        razreda je ta lastnost

    type = getFieldtype(returnType); //poskusi določiti tip atributa
        //za enostaven Javin tip. Če ne gre za
        //enostaven tip, metoda vrne null.

    if (type == null) {
        //očitno gre za kompleksen tip, oziroma entitetni razred ali
        //seznam le-teh. Ugotovimo za kakšne vrste povezavo gre
        //npr. many-to-many, one-to-one, itd...
        Če gre za „many-to-many“ ali „one-to-many“ {
```

```

        type = FieldType.LIST;
        subType = ugotovi ime ciljnega entitetnega tipa
    }

    Če gre za „many-to-one“ ali „one-to-one“ {
        type = FieldType.SUB_ASSET; //prikazan bo padajoči meni
        subType = ugotovi ime ciljnega entitetnega tipa
    }

    Če gre za tip Map {
        type = FieldType.MAP;
    }
}

Določi še nekatere dodatne lastnosti meta opisa, ki so potrebne,
če ne gre za enostaven tip (naprimer tip ključa in vrednosti če gre za MAP).

Ustvari nov MetaField in mu nastavi lastnosti, ki smo jih uspeli razbrati.
Vrni novo ustvarjeni MetaField.
}

```

Psevdo koda obeh metod je zelo poenostavljena. Popolna izvorna koda s komentarji je na voljo v *Prilogi C*.

3.2 Pretvorba entitete v Asset

Kot je že bilo povedano, strženik odjemalcu ne pošlje instance entitetnega razreda temveč instanco razreda *Asset*, ki vsebuje opise in vrednosti atributov. Pretvorbo iz entitete v *Asset* po potrebi izvaja servis *EditorService*.

Metoda *Asset getAsset(Object entity)* kot argument vzame instanco entitetnega razreda in ga pretvori v *Asset*. Pri pretvorbi se ponovno poslužuje Java Reflection API-ja, s katerim si pomaga iz podanega objekta (entitete) prebrati vrednosti persistenčnih lastnosti (entitetnih atributov). Metoda, ki vrača vrednost neke lastnosti nekega objekta, kot argument vzame ime te lastnosti, tako da je najprej potrebno priskrbeti seznam s temi imeni.

1. Pridobitev meta opisa entitetnega tipa

Ker je, kot že vemo, ime meta opisa entitetnega razreda enako imenu entitetnega razreda samega, lahko iz seznama meta opisov ustreznega poiščemo na podlagi imena razreda:

```

String type = entity.getClass().getSimpleName();
MetaAsset ma = core.getAssetDef(type)

```

2. Branje vrednosti persistenčnih lastnosti in iteracija čez attribute

Objekt *entity* najprej „ovijemo“ v *BeanMap*, ki v stilu razpršene tabele izpostavi lastnosti objekta in njihove vrednosti (ključ je ime lastnosti). Za implementacijo *BeanMap*-a je uporabljen Java Reflecton API, programerju pa poenostavi življenje, ker se ni potrebno ukvarjati z bolj kompleksnim Java Reflection API-jem, ko so potrebna le imena in vrednosti lastnosti objekta:

```
BeanMap map = new BeanMap(entity);
```

3. Nastavitev osnovnih lastnosti

Kreiramo novo instanco `asset`a in nastavimo vrednost primarnega ključa, naslova ter tipa:

```
Asset result = new Asset();
Serializable id = (Serializable) (map.get(ma.getIdField().getName()));
result.setId(id);
result.setType(type);

Object t = map.get(ma.getTitleField().getName());
result.setTitle(t != null ? t.toString() : "<no title>");
```

4. Procesiranje posameznega atributa

Sedaj lahko za vsako persistenčno lastnost (oziroma atribut) dobimo njeno vrednost:

```
for (MetaField mf : ma.getFields()) { //iteracija čez vse meta opise atributov.
    String fieldName = mf.getName(); //ime je enako imenu lastnosti entit. razr.
    Object obj = map.get(fieldName); //vrne vrednost lastnosti s podanim imenom.

    //nadaljujemo z obdelavo trenutnega atributa...
}
```

4.A Enostavni tipi

Enostavne lastnosti, kot so številke, nizi, datumi, itd... le dodamo v razpršeno tabelo, ki jo interno vsebuje `Asset`:

```
result.set(fieldName, obj);
```

4.B Entitete

Kadar gre za povezave med entitetami in je zato vrednost lastnosti dejansko objekt (ciljna entiteta) ali celo seznam entitet, shranimo samo vrednost primarnega ključa, naslovnega polja ter tip ciljne entitete. Rekurzivnega pretvarjanja v `Asset` torej ne izvajamo.

```
//najprej poišči vrednosti za primarni ključ, naslovno polje in tip ciljne entitete.
...
HashMap<String, String> subAsset = new HashMap<String, String>
subAsset.set("id", subAssetId); //
subAsset.set("title", subAssetTitle);
subAsset.set("type", subAssetType);
result.set(fieldName, subAsset);
```

Postopek je v tem opisu seveda poenostavljen in služi podajanju ideje algoritma. Izvorna koda s komentarji se nahaja v *Prilogi E*, kjer si bralec lahko pogleda, kako točno so v resnici obravnavani kompleksnejši primeri, kot na primer atributi tipa `Map` in podobno.

Pretvorba iz `Asset`-a v entiteto je podobna, le da se izvaja v obratni smeri (npr.: če se je pri opisnem algoritmu bralo iz `BeanMap` objekta, se tukaj piše vanj).

3.3 Generiranje urejevalnika

Za razliko od prejšnjih dveh pomembnih postopkov se generiranje urejevalnika dogaja na odjemalcu. Ko uporabnik želi začeti z urejanjem neke entitete, odjemalec od strežnika prejme objekt razreda *Asset*. Odjemalec mora nato prebrati seznam meta opisov entitetnih atributov in za vsakega kreirati ustrezno vnosno polje ter slednje dodati na vnosno formo. Ko so vsa polja pripravljena, na njih nastavi še vrednosti, ki jih prav tako prebere iz prejetega *Asset*-a.

V konstruktorju generičnega urejevalnika se najprej ustvari *FormPanel*, na katerega se nato dodaja posamezna polja. Nazadnje se polja poveže z vrednostmi iz *Asset*-a.

```
public GenericEditor(final Asset asset) {
    FormPanel panel = new FormPanel();
    . . . //inicializacija panela, itd...
    for (final MetaField f : asset.getFields()) {
        addField(asset, f, panel);
    }
    . . . //polja se zapolni z vrednostmi atributov iz asset-a
        //in se poskrbi za preostale elemente grafičnega vmesnika urejevalnika
        //kot je na primer gumb shrani, itd...
}
```

Večino dela z ustvarjanjem polj se opravi v metodi *addField*. Nekoliko več podrobnosti v zvezi z implementacijo je na voljo v *Prilogi G*. V prilogi je tudi seznam, ki določa, kakšno polje ustreza kateremu tipu atributa. V splošnem pa gre za naslednjo logiko:

```
if (f.getType() == FieldType.INTEGER || f.getType() == FieldType.LONG) {
    NumberField field = createIntegerField(f);
    panel.add(field);
} else if (f.getType() == FieldType.FLOAT) {
    NumberField field = createFloatField(f);
    panel.add(field);
} else if (f.getType() == FieldType.DATE) {
    DateField field = createDateField(f);
    panel.add(field);
} else if . . .
```

Poleg generičnega urejevalnika, obstajata še urejevalnik prodajnih artiklov in urejevalnik za binarne datoteke ter slike. Prvi se od generičnega pravzaprav razlikuje v tem, da ne doda vseh polj na eno stran, temveč ima več zavihkov, med katere razporedi polja. Tako so na prvem zavihku sami tista vnosna polja, ki so najpomembnejša in se jih najpogosteje ureja, itd... Za generiranje in dodajanje polj na posamezen zavihek še vedno uporablja isti algoritem kot generični urejevalnik. Oba prilagojena urejevalnika sta nekoliko podrobneje opisana v *Prilogi G*.

4. Konfiguracija in namestitvev

4.1 Dodatna konfiguracija z anotacijami

Z anotacijami na entitetnih razredih lahko povozimo privzete nastavitve oziroma specificiramo razne dodatne možnosti. Pogledali si bomo, katere anotacije so na voljo in čemu so namenjene. Te anotacije se pregledajo in uporabijo pri ustvarjanju meta opisov *MetaAsset* in *MetaField*. V prejšnjem poglavju jih pri opisu nismo omenjali zaradi splošnosti in razumljivosti postopka, je pa uporaba teh anotacij razvidna v izvorni kodi v *Prilogi C*.

4.1.1 CMSAsset

S to anotacijo lahko nekemu entitetnemu tipu določimo uporabniku prijazno ime (*label*) in specificiramo ali ga sploh vključimo v urejanje (*editable*). Vrednost napisa se med ustvarjanjem meta opisa tega tipa shrani v *MetaAsset.caption*. Če tipa ne vključimo v urejanje, ga ne bo možno najti niti z naprednim iskanjem in bo popolnoma skrit končnemu uporabniku.

```
@Target({ElementType.TYPE}) //anotiramo lahko nek razred
@Retention(RetentionPolicy.RUNTIME) //anotacija je prisotna tudi med tekom programa
public @interface CMSAsset {
    String label(); //napis za ta entitetni tip oziroma tip vsebine
    boolean editable() default true; //ali je dovoljeno urejanje?
}
```

Definicija anotacije CMSAsset

```
@Entity
@CMSAsset(label = "Sporočilo", editable = false)
public class Message {
    private String id;
    private User sender;
    . . .
}
```

Primer uporabe na entitetnem tipu Message:

Vsebinskemu tipu *Message* smo dali napis „Sporočilo“ in onemogočili njegovo urejanje v CMS-ju. Določanje napisa pri tipih, ki niso vključeni v urejanje, je sicer nesmiselno, ker se ga tako ali tako ne bo nikjer videlo...

4.1.2 CMSField

Konfiguriramo lahko tudi posamezne attribute in sicer lahko nastavimo napis, ki bo prikazan ob vnosnem polju v urejevalniku (*label*). Določimo lahko ali mora uporabnik obvezno vnesti vrednost (*required*), lahko pa atribut naredimo samo za branje (*readOnly*) ali ga celo popolnoma izločimo, da ne bo niti prikazan v urejevalniku (*editable*). V slednjem primeru se meta opis tega polja sploh ne zgenerira in se ga ne doda na seznam atributov na *MetaAsset*-u. V odjemalcu torej izgleda, kot da ta atribut oziroma polje sploh ne obstaja.. Pri nekaterih tipih atributov (naprimer pri tipu String) je treba določiti tudi želeno vrsto urejevalnika (*inputType*), saj iz samega tipa atributa ni možno ugotoviti kakšno polje naj se uporabi (pri String-u naprimer ne vemo ali gre za navaden znakovni niz ali pa za HTML, ki potrebuje HTML urejevalnik namesto preprostega vnosnega polja). V spodnji tabeli je podano, kateri parameter iz anotacije se med generiranjem meta opisa preslika v katero lastnost meta opisa *MetaField* (lastnosti in njihov pomen so sicer našteje oziroma obrazložene v *Prilogi B*).

Parameter	Lastnost MetaField	Parameter	Lastnost MetaField
label	caption	editable	/
order	order	readOnly	readOnly
required	required	inputType	inputType

```
@Target({METHOD, FIELD}) //anotiramo lahko bodisi lastnost razreda bodisi metodo (getter)
@Retention(RUNTIME)
public @interface CMSField {
    String label() default "no title"; //napis polja
    int order() default 100; //vrstni red
    boolean required() default false; //ali je vrednost obvezno treba vnesti
    boolean editable() default true; //ali je polje sploh prikazano v urejevalniku
    boolean readOnly() default false; //ali je vrednost samo za branje
    InputType inputType() default InputType.GUESS; //kakšne vrste naj bo vnosno polje
}
```

Definicija anotacije CMSField

4.1.3 TitleField

TitleField nima nobenih parametrov. Njegov namen je preprost in sicer z njim označimo atribut, ki naj se uporabi kot naslov vsebine. Vrednost tega atributa se v odjemalcu prikazuje v tabelah in padajočih menijih za izbor, itd... Ponavadi za naslovni atribut izberemo takega, za katerega vemo, da bo učinkovito povzel posamezno instanco tega tipa vsebine. Pri člankih je to naprimer naslov članka, pri prodajnih artiklih je to ime artikla (npr. Acer Aspire 5920G, itd...). Če tega polja na entitetnem tipu ne specificiramo, CMS sam izbere prvi atribut, ki ga sreča pri analizi entitetnega razreda, kar je posledično lahko zelo nepregledno oziroma neintuitivno v uporabniškem vmesniku.

```
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TitleField {
}
```

Definicija anotacije CMSField

4.2 Namestitev

Za namestitev potrebujemo aplikacijski strežnik JBoss AS 5. Z nekaj prilagoditvami bi se sicer dalo Parsek CMS 2.0 uporabljati na praktično katerem koli J2EE strežniku, vendar za to trenutno ni niti potrebe niti želje.

Namestitev je najenostavnejša, če se Parsek CMS 2.0 namesti kar na strežnik od ciljne aplikacije, čeprav je možna namestitev tudi brez tega. V nadaljevanju si bomo pogledali obe možnosti.

4.2.1 Skupaj s ciljno aplikacijo

Potrebno je le pripraviti ustrezno konfiguracijo persistenčne enote v temu namenjeni standardni XML datoteki *persistence.xml*. Ta konfiguracija je del vsake aplikacije ki za persistenco uporablja JPA.

Tukaj sta izpostavljena najpomembnejša dva vnosa, ki CMS-ju zagotavljata dostop do entitet ciljne aplikacije in sicer izbira *EntityManagerFactory*-ja ter podatkovnega vira.

Ime za *EntityManagerFactory* podamo z naslednjo vrstico:

```
<property name="jboss.entity.manager.factory.jndi.name" value="java:/MyAppEntityManagerFactory" />
```

Ime podatkovnega vira pa s sledečo:

```
<jta-data-source>java:/MyAppDS</jta-data-source>
```

Z ustreznim imenom *EntityManagerFactory*-ja zagotovimo, da so CMS-ju na voljo **entitetni razredi** ciljne aplikacije, z izbiro imena podatkovnega vira pa določimo, katero **podatkovno bazo** naj CMS uporablja. Vsak podatkovni vir ima namreč svoje ime in konfiguracijo povezave s podatkovno bazo, ki se nahaja v posebni XML datoteki na aplikacijskem strežniku in je ni treba posebej izdelati, če CMS namestimo na isti strežnik, kjer je že nameščena tudi ciljna aplikacija, saj je bilo tovrstno konfiguracijo potrebno pripraviti že zanj, CMS pa jo bo uporabil.

Ko smo poskrbeli za konfiguracijo, nam preostane samo še da Parsek CMS 2.0 prevedemo in zapakiramo v spletni arhiv WAR (web archive), ki je standardna datoteka za distribucijo Javinih spletnih aplikacij. WAR datoteko nato skopiramo v ustrezno mesto na aplikacijskem strežniku in namestitev je končana (razen uporabniških imen in gesel, ki se jih ureja kar naknadno iz samega Parsek CMS 2.0 sistema).

4.2.2 Samostojna namestitve

Če Parsek CMS 2.0 namestimo na ločen strežnik ali pa celo urejamo podatke aplikacije, ki ne uporablja JPA za persistenco, je vmes potreben še dodaten korak, saj v teh primerih nimamo že obstoječega *EntityManagerFactory*-ja. Tarkat je treba uporabiti Hibernate-ovo orodje „hbm2java“, ki se poveže na ciljno podatkovno bazo in iz tamkajšnjega podatkovnega modela zgenerira ustrezne entitetne razrede. Slednje dodamo v distribucijsko WAR datoteko in vsakega od entitetnih razredov še navedemo v datoteki *orm.xml*, ki je prav tako standardna konfiguracijska datoteka za JPA. Sicer pa je postopek skoraj enak kot v prvem primeru.

Možnost samostojne namestitve nam torej omogoča urejati podatke iz praktično katerekoli podatkovne baze, s čimer je Parsek CMS 2.0 zelo univerzalno in „nevsiljivo“ orodje, saj spremembe obstoječe infrastrukture ali aplikacij niso potrebne – z izjemo tega, da potrebuje ustrezen aplikacijski strežnik, ki pa se ga lahko namesti tudi v virtualno okolje.

5. Obstoječi sistemi

Ob pregledu obstoječih sistemov za upravljanje vsebin, predvsem tistih, ki so bili razviti oziroma se uporabljajo v našem podjetju, je mogoče razbrati nekatere bistvene lastnosti, ki so skupne vsem in kot take torej predstavljajo dobro osnovo, na kateri bi bilo smiselno graditi tudi novo rešitev, kot nekakšno nadgradnjo oziroma prilagoditev obstoječih. Tak pristop prinaša dve pomembni prednosti, ki ju ne gre spregledati in sta pravzaprav ključnega pomena.

Kot prvo, se s tem lahko ohranijo praktično preizkušene funkcionalnosti in ideje, ki so se med dolgoletno uporabo pri nas oziroma pri naših naročnikih že dobro izkazale. Kot take jih torej kaže preučiti, po potrebi nekoliko prilagoditi in jih vključiti tudi v nov sistem ter se tako v veliki meri izogniti tveganju, da bi bil le-ta zgrešen že v svojem bistvu, funkcionalne nadgradnje pa tako ne bi prišle do pravega izraza.

Kot drugo, pa je na tak način lažje zagotoviti združljivost z obstoječimi rešitvami in tako olajšati morebiten prehod iz njih na nov sistem. Na združljivost moramo ob tem gledati predvsem iz dveh zornih kotov in sicer iz vidika podatkovnega modela, kjer je zaželeno, da bi le-ta ostal enak oziroma popolnoma kompatibilen, s čimer se neposredno reši prvi problem pri zamenjavi sistema - obstoječe spletne aplikacije lahko ostanejo nespremenjene. Za gladek prehod pa je potrebno poskrbeti tudi za uvajanje uporabnikov, kar je veliko lažje in hitrejšo, če se v novem sistemu že na začetku počutijo domače, ker so primeri uporabe in organiziranost vsebin podobni, kot so jih že vajeni. Tako se hitro privadijo na spremembe, pri uvajanju pa se raje bolj posvečajo novim funkcionalnostim. Iz drugega vidika torej ciljamo predvsem na „združljivo“ uporabniško izkušnjo.

Največji vzornik za nov sistem za upravljanje vsebin je bi Parsek CMC (Content Management Console). Le-ta je ena izmed komponent celovite platforme CP2, na kateri so včasih temeljile naše spletne aplikacije. Kot takega sem ga izbral za eno od obstoječih rešitev, ki jo bom opisal in primerjal s Parsek CMS 2.0

Ker konkurenčnih rešitev v pravem pomenu besede praktično ni, saj se nobena ne osredotoča na urejanje poslovnih objektov in prilagajanju ciljni aplikaciji, bo sledil opis enega od tipičnih sistemov CMS, ki predpostavlja svoj podatkovni model in uporabo svojega sistema predlog. Opisan bo sistem Magnolia CMS. Primerjava Magnolie s Parsek CMS 2.0 pa bo z vsemi razlikami še bolj izpostavila posebnost našega novega sistema za upravljanje z vsebinami.

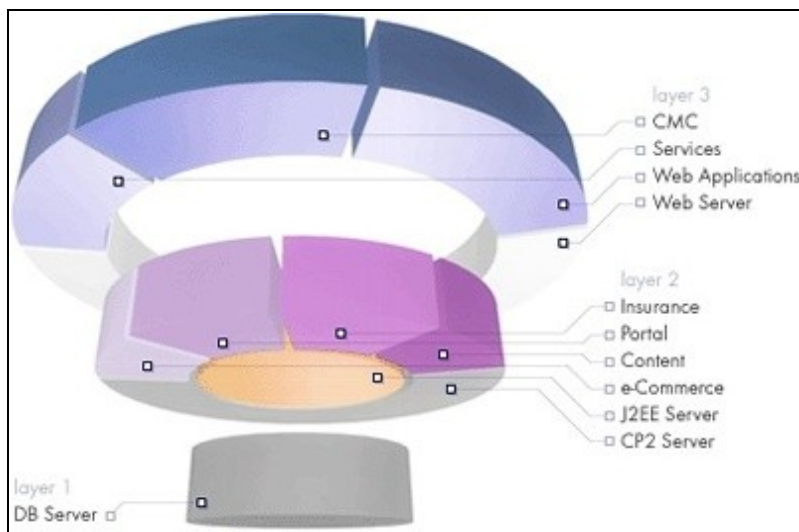
Nazadnje bo opisano še ogrodje Metawidget, ki sicer ni sistem CMS, je pa ogrodje, ki bi ga lahko uporabil za izdelavo Parsek CMS 2.0 in sicer je namenjeno samodejnemu generiranju uporabniških vmesnikov. Pogledali si bomo, kakšne predelave in kompromisi bi bili potrebni, če bi namesto lastne implementacije nekaterih funkcionalnosti, uporabil omenjeno ogrodje.

5.1 Parsek CMC

5.1.1 Tehnologija Parsek CP2

Parsek CP2 (Parsek Content Platform 2) je lastna tehnologija, s katero smo včasih razvijali naše spletne aplikacije, sedaj pa se opušta na račun naprednih odprto-kodnih tehnologij, ki omogočajo podobne funkcionalnosti, vendar včasih niso bile na voljo, oziroma zrele za uporabo v produkciji.

CP2 ima tri-nivojsko arhitekturo, ki na najnižjem nivoju zajema podatkovno bazo. Na srednjem nivoju je aplikacijski strežnik z različnimi namenski moduli in tako imenovani CP2 strežnik, ki razširja J2EE strežnik in razvijalcem daje na voljo orodja ter funkcionalnosti, ki jih ni v sami J2EE specifikaciji. Primer malo preprostejšega modula je modul za vsebine,

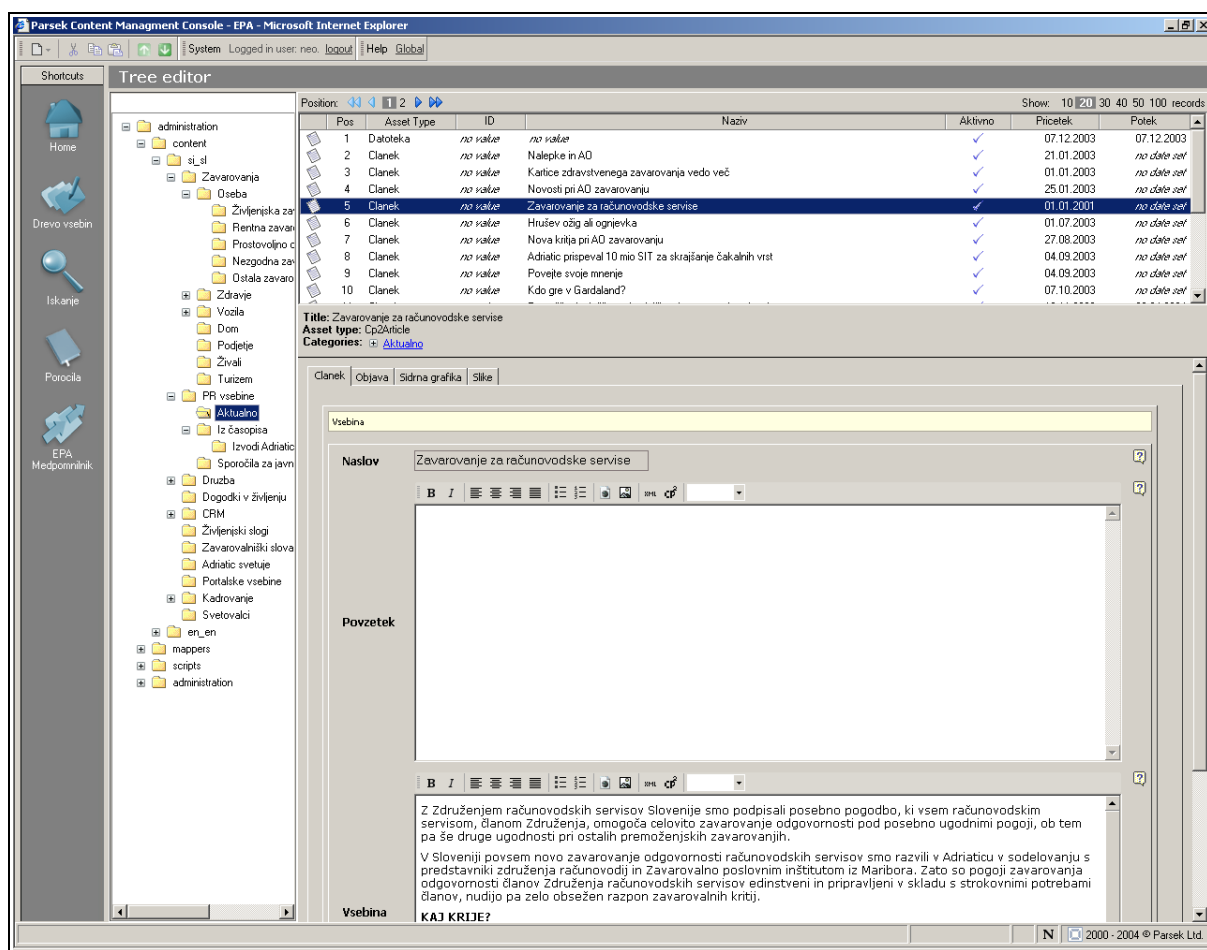


kot so članki, slike, naložene datoteke, itd..., ki jih, ustrezno transformirane, posreduje predstavitvenemu nivoju, povezuje pa se tudi s sistemom za urejanje vsebin. Kot primer kompleksnejšega modula lahko vzamemo zavarovalniški modul, ki implementira poslovno logiko, značilno za zavarovalniške spletne aplikacije – od informativnih izračunov do spletnega sklepanja zavarovanj tako za zavarovalniške agente kot za stranke same. Tudi tukaj pride v poštev urejanje vsebin – izgled vnosnih form, višine premij, itd... Najvišji nivo pa vsebuje komponente, ki so specifične za posamezno spletno aplikacijo, torej uporabniške vmesnike (ponavadi v obliki spletnih strani) in kontrolne razrede, ki stojijo za njimi ter specifične storitve, naprimer za integracijo z zunanjimi sistemi. Na najvišjem nivoju so tudi nekatere komponente, ki so del vsakega CP2 sistema in niso specifične za posamezno rešitev – tak primer je tudi spletni vmesnik CMC za upravljanje z vsebinami.

5.1.2 Uporabniški vmesnik

CMC ima spletni uporabniški vmesnik, ki deluje v Internet Explorer-ju 5.5 ali novejšem, ne podpira pa ostalih brskalnikov. Danes lahko tako omejenost smatramo za resno pomanjkljivost, a v času razvoja z alternativami ni bilo mogoče razviti dovolj bogatega spletnega uporabniškega vmesnika, ki bi solidno deloval, oziroma so bile razlike med brskalniki prevelike in stroški njihove podpore zato neracionalni. Ker je CMC orodje, ki se uporablja samo znotraj kontroliranega okolja (vnašalci vsebin v našem podjetju ali pri naročniku), to v resnici ni predstavljalo resnega problema. Zaradi večje razširjenosti ostalih brskalnikov ter prodora Linuxa in drugih odprto-kodnih projektov v podjetja, pa je postajala

ta omejitev vedno bolj problematična. Nezanemarljiva je postala tudi razširjenost Mac OS-a.



Slika 9: Drevesni urejevalnik

Modul „Tree editor“, ki na levi prikazuje drevo kategorij, v katere so umeščeni posamezni gradniki vsebine. Zgornja tabela prikazuje vsebino izbrane kategorije, v spodnjem delu pa je odprt urejevalnik izbranega gradnika vsebine, v tem primeru je to urejevalnik članka.

Podjetja, pri katerih celotna informatika temelji na prosti programski opremi, ponavadi kot enega izmed pogojev za sodelovanje postavijo ravno podporo odprtih tehnologij. Podpora večine modernih brskalnikov in namiznih operacijskih sistemov je tako postala ena od glavnih zahtev in vodil pri razvoju novega sistema predvsem pri izbiri tehnologij, na katerih bo razvoj potekal.

5.1.3 Še nekatere ključne lastnosti

Organiziranost vsebin

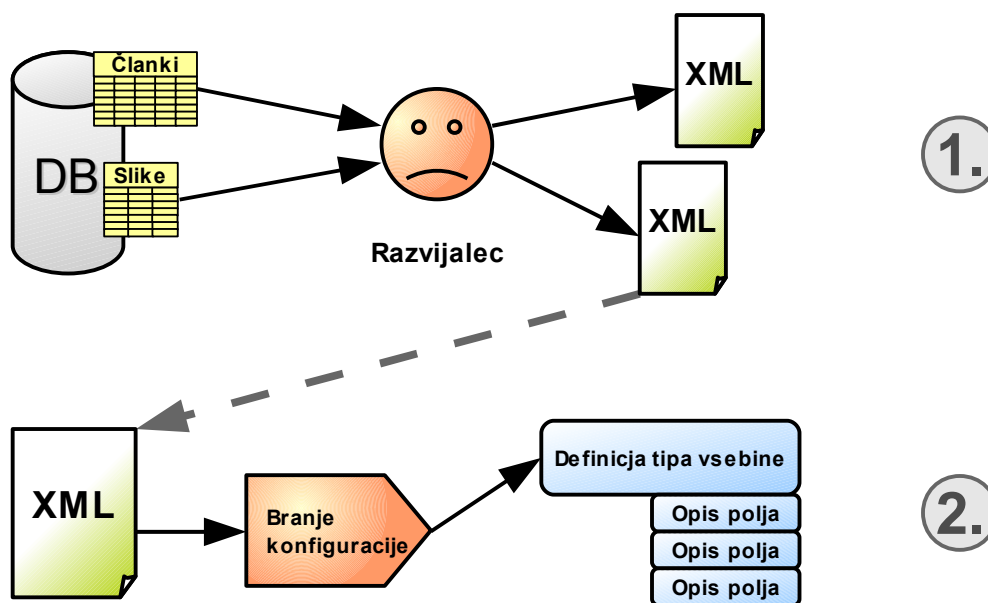
Tudi v Parsek CMC oziroma kar celotni CP2 platformi so vsebine organizirane v drevesno

strukturo, kot bi lahko že sklepali iz zaslonske slike. Ta princip je praktično enak kot v Parsek CMS 2.0 in je tudi realiziran z enakim podatkovnim modelom.

Neodvisnost od tipa vsebin

CMC sicer omogoča urejanje poljubnih vsebin, vendar ne tako enostavno kot Parsek CMS 2.0. Vsak entitetni tip, ki ga želimo urejati, je potrebno opisati v posebni konfiguracyjski XML datoteki, v kateri navedemo njegove lastnosti in kakšna naj bodo vnosna polja za urejanje. Na podlagi teh informacij je mogoče dinamično generirati vnosne forme, za kar se uporablja lastna implementacija XForms standarda. Entitetni tip, pa je potrebno definirati še za Hibernate (v prvi XML datoteki so zgolj napotki za Parsek CMC) zopet v XML datoteki, tako da je na voljo CP2 platformi. Še ena neprijetna omejitev starega sistema, ki izhaja iz integriranosti Parsek CMC v CP2 in sicer, da je zmožen urejati samo entitete tistih aplikacij, ki so zgrajene s tehnologijo CP2.

XML konfiguracija pa kaj hitro postane preobsežna in zato golemu očesu nepregledna (slednje je sicer že v splošnem znan problem uporabe jezika XML), še posebej pri kompleksnejših tipih in ko je le-teh veliko. Tako smo prišli do naslednje pomembne zahteve za nov urejevalnik vsebin – enostavnejši in preglednejši način konfiguracije in definiranja vsebinskih tipov ter možnost urejanja neglede na tehnologijo ciljne aplikacije.



Slika 10: Uvedba novega vsebinskega tipa

- (1.) Razvijalec pripravi opise v obliki XML in jih vključi v konfiguracijsko XML datoteko.
- (2.) CMC ob zagonu slednjo prebere in shrani informacije o tipu vsebine v temu namenjen objekt. V spominu vzdrži seznam vseh možnih tipov, ki so se nahajali v konfiguracijski datoteki.

```

<asset name="Cp2Image">
  <caption>Image</caption>
  <category-templates>
    <category-template category-id="7" asset-id="2"/>
    <category-template category-id="11" asset-id="20"/>
  </category-templates>
  <fields title-field="Title">
    <step id="1">
      <caption>Basic data</caption>
      <page-caption>Image</page-caption>
      <group id="a" caption="test">
        <field read-only="false" required="true">
          <name>Title</name>
          <type>string</type>
          <input-type name="input"/>
          <label>Image Title</label>
          <bind-to module-name="datagrid">title</bind-to>
        </field>
        <field read-only="false" required="true" searchable="true">
          <name>PublishDate</name>
          <type>date</type>
          <input-type name="input"/>
          <label>PublishDate</label>
          <hint>From when is this article published</hint>
          <bind-to module-name="_listgrid">publishedDate</bind-to>
          <bind-to module-name="datagrid">publishDate</bind-to>
        </field>
        <field read-only="false" required="false">
          <name>ExpiryDate</name>
          <type>date</type>
          <input-type name="input"/>
          <label>ExpiryDate</label>
          <hint>When does article expires</hint>
          <bind-to module-name="datagrid">expDate</bind-to>
        </field>
        <field read-only="false" required="false">
          <bind-to module-name="datagrid">published</bind-to>
          <bind-to module-name="_listgrid">published</bind-to>
          <name>Published</name>
          <type>boolean</type>
          <input-type name="input"/>
          <label>Published</label>
          <hint>Is this article published?</hint>
        </field>
      </group>
    </step>
    <step id="2">
      <caption>Image</caption>
      <page-caption>Select Image</page-caption>
      <group id="image">
        <field read-only="false" required="true">
          <name>Bytes</name>
          <type>binary</type>
          <input-type name="image"/>
          <label>Image</label>
          <hint>Please select image!</hint>
        </field>
      </group>
    </step>
  </fields>
</asset>

```

Primer definicije tipa vsebine:

Prikazan je del konfiguracije, ki definira slikovno datoteko. Ta vsebinski tip je razmeroma preprost, a je definicija kljub temu že dokaj obsežna. Želimo si preglednejši in preprostejši način!

Neodvisnost od SUPB

CP2 za delo s podatkovno bazo uporablja ORM ogrodje Hibernate, ki podpira raznovrstne SUPB-je. V prvi uporabljeni verziji Hibernate-a, verziji 0.6, je bil slednji še precej v povojih, tako da ga je bilo treba razširiti z različnimi skriptami in podobno. Tudi definicija entitet je bila nerodna, ker se je to počelo v XML datotekah. Tako je bilo, kot smo že omenili, ob uvedbi novega entitetnega tipa najprej bilo treba poskrbeti za samo definicijo entitete za Hibernate v eni XML datoteki, nato pa še v konfiguraciji CMC-ja dodati ustrezen opis, če jo je bilo treba urejati.

5.1.4 Primerjava

Lastnost, ki najbolj bode v oči je, da je Parsek CMC integriran s CP2 in ga tako ne moremo uporabiti za urejanje, če ne namestimo celotne CP2 platforme, kar pa je nekoliko nerodno in tudi potrata strežniških resursov. To sicer pri aplikacijah, ki so narejene na CP2 ni problem, ker je slednji že nameščen, vseeno pa si želimo bolj neodvisen in „lahek“ sistem za upravljanje z vsebinami in ravno to Parsek CMS 2.0 tudi je.

V primerjavi s Parsek CMS 2.0, je pri Parsek CMC za uvedbo nekega entitetnega tipa postopek bolj zapleten. Entitetni tip je potrebno dvakrat konfigurirati. Prvi XML je konfiguracija za ORM ogrodje Hibernate in CP2 platformo, drugi XML pa za sam Parsek CMC. Tudi Parsek CMS 2.0 uporablja Hibernate, vendar uporablja novejšo verzijo in XML konfiguracija ni potrebna ker jo nadomeščajo anotacije na entitetnih razredih, ki so bolj pregledne. Dodatna konfiguracija za sam Parsek CMS 2.0 je opsijska, med tem ko Parsek CMC brez svoje konfiguracije za nek entitetni tip slednje ne bo zmožen urejati.

Oba sistema sicer uporabljata spletni vmesnik, vendar je Parsek CMC omejen zgolj na Internet Explorer, med tem ko Parsek CMS 2.0 podpira vse brskalnike, ki jih podpira tudi GWT. Uporabniška vmensika sta si podobna v drevesu kategorij in seznamu v kategorijo povezanih vsebin, med tem ko je največja novost (in prednost) v Parsek CMS 2.0 uporaba zavihkov, kar omogoča hkratno urejanje večih vsebin znotraj enega okna brskalnika.

5.1.5 Možnost migracije na Parsek CMS 2.0

Ob analizi možnosti migracije ugotovimo, da pravzaprav skoraj ni nobenih ovir. Entitetni tipi za realizacijo drevesa kategorij in za shranjevanje osnovnih tipov vsebin so ostali praktično enaki. Potrebno je torej samo narediti entitetne razrede z orodjem hbm2java (za nestandardne tipe vsebin oziroma za ostale poslovne objekte) in Parsek CMS 2.0 skupaj z njimi namestiti na ustrezen strežnik. Po potrebi seveda lahko entitetne razrede pred tem anotiramo z že omenjenimi anotacijami za dodatno konfiguracijo CMS-ja.

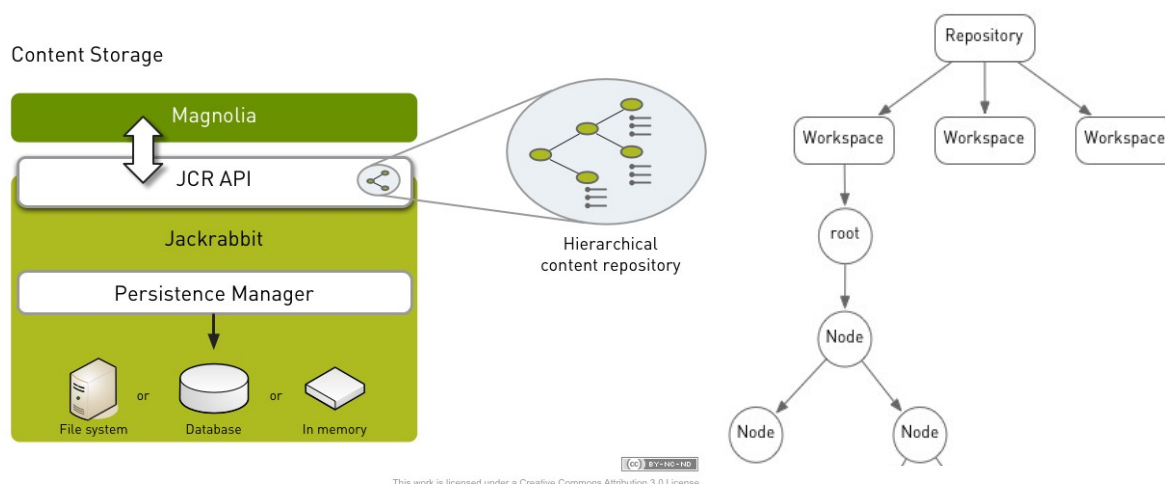
5.2 Magnolia CMS

Magnolia CMS je sistem za upravljanje z vsebinami, ki se primarno osredotoča na urejanje vsebin spletnih mest oziroma portalov in hitro postavitev le teh. Obstaja v dveh različicah (obe sta oprto-kodni) in sicer kot komercialna varianta „Enterprise Edition“ in brezplačna varianta „Community Edition“. Tukajšnji opis drži za obe različici.

5.2.1 Content repository

Magnolia za razliko od večine sistemov CMS ne dela s podatkovno bazo ali entitetami temveč vsebine shranjuje v takoimenovan repozitorij vsebine („*content repository*“) preko standardnega vmesnika JCR („*Java Content Repository API*“). Podobno kot aplikacije z uporabo vmesnika JPA dosežejo neodvisnost od SUPB in tudi od implementacije samega vmesnika, tako je Magnolia z uporabo vmesnika JCR neodvisna od same implementacije repozitorija vsebine, privzeto pa se sicer uporablja Apache-jeva implementacija Apache Jackrabbit.

Za JCR-jem stoji hierarhična podatkovna struktura. Gre za drevo vozlišč, kjer vsakemu vozlišču pripadajo lastnosti, v katerih so shranjeni podatki. Slednji so lahko tekstovni nizi, številke, datumi ali poljubno veliki binarni podatki, obstaja pa še nekaj posebnih tipov. Tudi vozlišča so različnih tipov, poleg običajnih obstajajo naprimer t.i. vozlišča z referenco, ki jih uporabnik lahko direktno naslovi preko njihove reference (ni potrebno potovanje po drevesu) ali pa vozlišča z verzioniranjem, ki hranijo zgodovino sprememb, kar je pri sistemih za upravljanje vsebin še posebej praktično. V repozitoriju je več delovnih okolij (*workspace*), vsako delovno okolje pa vsebuje svoje drevo. JCR podpira tudi transakcije, opazovanje sprememb (ob spremembi obvesti zainteresirane komponente), uvoz in izvoz XML-ja (hierarhija v JCR in XML se naravno ujameta, saj je tudi XML hierarhično urejen jezik značk, atributov in vrednosti), avtentifikacijo in avtorizacijo, iskanje z XPath ali SQL poizvedbami, predpomnenje, itd... Nekatere funkcionalnosti so po JCR specifikaciji sicer neobvezne (npr. podpora poizvedbenega jezika SQL).



Slika 11: Magnolia, Jackrabbit in repozitorij vsebine

Uporaba JCR-ja prinese razvijalcem sistemov CMS pomembno prednost in sicer je za hierarhično urejenost vsebin poskrbljeno že v osnovi s samim konceptom hierarhije v repozitoriju. V Parsek CMS 2.0 pa je bilo treba to posebej implementirati in sicer s sedaj bralcu že znanim drevesom kategorij.

5.2.2 Urejanje vsebin

The screenshot shows the Magnolia CMS interface. On the left is a sidebar with navigation links: Website, Documents, Forums, Inbox, Data, Templating Kit, Security, Configuration, Tools, Packager, and Magnolia Store. The main area displays a tree view of the content hierarchy under 'demo-project'. A context menu is open over the 'about' page, showing options like 'Open page...', 'New page', 'Delete page', 'Move page', 'Copy page', 'Versions', 'Activate this page', 'Activate incl. sub pages', 'De-activate', 'Export tree to xml', and 'Import from xml'. Below the tree view is a table of pages.

Page	Title	Status	Template	Mod. date
demo-project	Home	●	Home	10-03-12, 18:13
about	About	●	Section	09-11-12, 16:17
subsection-article	Articles	●	Section	09-11-12, 16:17
article	Standard Article	●	Article	10-09-06, 17:21
large-article	Article with TOC	●	Large Article	-
an-interesting	Interesting Article	●	Article	-
history	History	●	Article	09-11-12, 16:03
news-and-events	News and Events	●	Section	10-03-10, 15:38
multimedia	Multimedia	●	Section	10-03-10, 15:05
service	Service	●	Section	09-11-12, 16:55
members-area	Members Area	●	Section	10-03-15, 12:01
demo-features	Home	●	Home	09-11-19, 15:54

At the bottom of the interface, there are buttons for 'Open page...', 'New page', 'Activate this page', 'De-activate', and 'Refresh'.

Slika 12: Hierarhija vsebine spletnega mesta

Na levi je izbira modula, na desni pa njegova zaslonska maska, podobno kot pri Parsek CMC. Hierarhija iz tukaj prikazanega drevesa sicer sovпада z vozlišči v JCR-ju, vendar pa to ni pregledovalnik JCR-ja, pri slednjem namreč vidimo tudi lastnosti posameznih vozlišč in podobno. Ta pogled je namenjen pregledovanju strukture spletnega mesta.

The screenshot shows the 'About' page in the Magnolia CMS. The top navigation bar includes links for Home, About (active), News and Events, Multimedia, Service, and Members Area. Below the navigation bar is a banner section with the text: 'This is the banner text for the "About" section. It is inherited by all child pages unless these are configured differently. A banner can be disabled in the template configuration settings.' The main content area is divided into several sections: 'Articles' (with a sub-section 'History'), 'About' (with a sub-section 'Interesting Article'), 'Events' (with a sub-section 'New Year's Eve'), and 'Look here' (with a sub-section 'More Interesting news'). Each section has a corresponding image and text. The 'About' section includes a sub-section 'Interesting Article' with a sub-section 'Occidental in fact' and a sub-section 'Interesting Article'.

Slika 13: Urejanje vsebine

Izbrano spletno stran za urejanje odpremo z „Open page...“ in le-ta se prikaže z dodanimi (zelenimi) orodnimi vrsticami z gumbi za različne operacije urejanja. Tak način urejanja je zelo intuitiven in se ga tudi uporabniki začetniki hitro privadijo.

The screenshot shows a web-based form for editing an article header. The form is titled 'Article Header' and 'Category'. It includes the following fields and controls:

- Headline (en) (*)**: Text input field containing 'Interesting Article'.
- Kicker (en)**: Text input field containing 'Occidental in fact'.
- Date**: Text input field containing '2008-11-18' and a 'Select date...' button.
- Author (en)**: Text input field containing 'B. Bauer'.
- Abstract (en)**: Text area containing a paragraph of text: 'This is the abstract for the article page "Interesting Article". It is a brief résumé on the content of this page. The abstract on the beginning of a page is also used in teasers and openers that reference this page. Li European lingues es membres del sam familie. Lor separat existentie es un myth. Por scientie, musica, sport etc, litot Europa usa li sam vocabular. Li lingues differe solmen in li grammatica, li pronunciation e li plu commun vocabules.'
- Image**: Section with two radio buttons: 'Upload' and 'Dms' (selected). Below them is a 'Choose Internal Image' field with the path '/demo-project/img/opener/green-chairs-...' and an 'Internal link...' button. Below that is a 'Choose an Image from documents' link.
- Image Location**: Dropdown menu with 'above the text' selected. Below it is a hint: 'Select where the image should be placed'.
- Buttons**: 'Save' and 'Cancel' buttons at the bottom right.

Slika 14: Dialog za urejanje

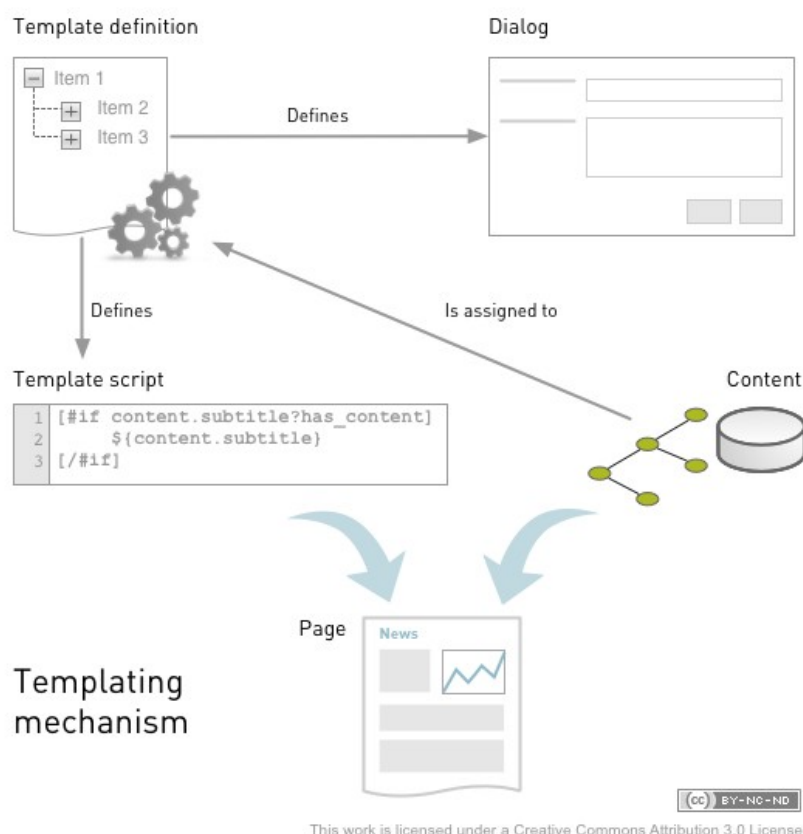
Po kliku na „Edit article header“ nas pričaka takšen dialog z vnosnimi polji za urejanje posameznih lastnosti.

Hierarhija iz modula Website direktno (lahko bi rekli „ena na ena“) sovpada s prikazano vsebino na spletnem mestu.

Problem se pojavi, ko bi želeli neko vsebino imeti vnešeno samo enkrat in prikazano na različnih straneh (lahko bi rekli „ena na več“). V tak primer uporabe spadajo tudi izdelki za spletno trgovino, kjer se nek točno določen izdelek lahko pojavi na različnih koncih, naprimer v različnih kategorijah izdelkov in podobno. Izdelek je strukturirana podatkovni tip in ga želimo hraniti neodvisno od predstavitvene logike, ki jo narekuje drevo vsebin spletnega mesta. Temu je namenjen modul Data, v katerem je možno definirati take podatkovne tipe. V njem imamo možnost definirati podatkovne tipe, ki nam predstavljajo poslovne objekte.

5.2.3 Standard Templating Kit

Magnolia CMS je celovita rešitev za izdelavo spletnih mest in kot taka vključuje tudi svoj sistem predlog imenovan *STK*. Vsaka posamezna izrisana spletna stran ali del strani (paragraf) je neka predloga, zapolnjena z vsebinami oziroma podatki iz repozitorija.



Slika 15: Mehanizem predlog

Definicija predloge je sestavljena iz različnih elementov, ki se jih nato lahko preko posebnih značk vstavlja v skripto predloge. Elementi se zapolnijo z vrednostmi iz repozitorija. Vsebino repozitorija se ureja preko dialogov. Na sami definiciji predloge je določeno, katera skripto predloge se uporablja za prikaz in kateri dialog se uporablja za urejanje vsebine. Predloga je lahko predloga celotne strani ali pa samo nekega dela (predloga paragrafa).

Magnolia za izrisovanje predlog privzeto podpira tehnologiji JSP in FreeMarker, lahko pa napišemo izrisovalnik tudi za kak drug jezik. V distribuciji Magnolie se nahajajo številne predloge in dialogi, ki običajno že zadovoljujejo potrebe za izdelavo spletnih strani in portalov, tako da se lahko razvoj osredotoči na samo končno podobo in na vsebino. Po potrebi je možno za naprednejše rešitve napisati svoje predloge in dialoge za urejanje.

Hierarchical content nodes in JCR

1	NBA	
2	Bulls-and-the-Thunder-Make-a-Stand	
	title	Bulls and the Thunder Make a Stand
	date	4/22/10
	author	The Associated Press
	abstract	Derrick Rose scored 31 points Thursday night, and
	hideInNav	false
	imageLocation	left
	navigationTitle	Bulls and the Thunder
3	main	
	0	
4	text_files	
	image	dms
	imageDmsUUID	6c433a89-ecb2-4df1-a995-f60109e95991
	imageLocation	right
	text	<p>LeBron James scored 13 of his 39 points
5	00	
	text_files	
	image	left
	subtitle	THUNDER 101, LAKERS 96
	text	<p>Kevin Durant had 29 points and 19 rebounds
6	footer	
	title	Sections
	copyright	Copyright 2010 The New York Times Company

Rendered article

1 Bulls and the Thunder Make a Stand

22.04.2010 The Associated Press [Print](#) [Bookmark](#)

Derrick Rose scored 31 points Thursday night, and the host Chicago Bulls built a 21-point lead before holding on against the top-seeded Cleveland Cavaliers, 108-106, in Game 3 of their first-round series. Cleveland leads the best-of-seven Eastern Conference series, 2-1.

2

LeBron James scored 13 of his 39 points in the fourth quarter, but had three turnovers down the stretch. Still, the Cavaliers pulled to 107-106 after Mo Williams's 3-pointer with 3.8 seconds left, and Anthony Parker missed a long 3 at the buzzer.

The Bulls led by 39-23 in the second quarter after a 3-point play by Luol Deng (20 points) capped an 18-5 run.

3

THUNDER 101, LAKERS 96

Kevin Durant had 29 points and 19 rebounds in the first playoff game in Oklahoma City, lifting the Thunder to a victory against Los Angeles.

Chicago Bulls guard Derrick Rose scored 31 points in the Bulls victory.
Tannen Maury/European Pressphoto Agency

4

Sections
[Home](#) | [World](#) | [U.S.](#) | [Technology](#) | [Science](#) | [Health](#) | [Sports](#)
 Copyright 2010 The New York Times Company

5

[CC BY-NC-ND](#)

This work is licensed under a Creative Commons Attribution 3.0 License

Slika 16: Odnos med vozlišči v JCR in spletno stranjo

Hierarhijo na levi prikazuje orodje JCR Browser in ne modul Website.

Page script

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
[#assign cms=JspTaglibs["cms-taglib"]]

<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  [#include "header.ftl"]
</head>

<body>
  [#cms.mainBar label="Page Info" dialog="pageInfo"/]
  <div id="main">
    [#cms.contentNodeIterator contentNodeCollectionName="main"]
    [#cms.includeTemplate /]
    [#cms.contentNodeIterator /]
    [#cms.newBar contentNodeCollectionName="main"
    paragraph="textImage" /]
  </div>
  <div id="footer">
    [#cms.editBar contentNodeName="footer" paragraph="footer" /]
    [#cms.includeTemplate contentNodeName="footer" /]
  </div>
</body>
</html>
```

Paragraph script

```
[#assign cms=JspTaglibs["cms-taglib"]]
[#cms.editBar /]
<h2>${content.title}</h2>

${content.text}
```

Paragraph script called
with includeTemplate tag

Rendered article

1 [« Preview](#) [AdminCentral](#) [Page Info](#)

Bulls and the Thunder Make a Stand

22.04.2010 The Associated Press [Print](#) [Bookmark](#)

Derrick Rose scored 31 points Thursday night, and the host Chicago Bulls built a 21-point lead before holding on against the top-seeded Cleveland Cavaliers, 108-106, in Game 3 of their first-round series. Cleveland leads the best-of-seven Eastern Conference series, 2-1.

2 [Edit](#) [Move](#) [Delete](#)

BULLS 108, CAVALIERS 106

LeBron James scored 13 of his 39 points in the fourth quarter, but had three turnovers down the stretch. Still, the Cavaliers pulled to 107-106 after Mo Williams's 3-pointer with 3.8 seconds left, and Anthony Parker missed a long 3 at the buzzer.

The Bulls led by 39-23 in the second quarter after a 3-point play by Luol Deng (20 points) capped an 18-5 run.

3 [New Content](#)

4 [Edit](#) [Move](#) [Delete](#)

Sections
[Home](#) | [World](#) | [U.S.](#) | [Technology](#) | [Science](#) | [Health](#) | [Sports](#)
 Copyright 2010 The New York Times Company

[CC BY-NC-ND](#)

This work is licensed under a Creative Commons Attribution 3.0 License

Slika 17: Odnos med skripto predloge za članek in izrisom članka.

5.2.4 Modul Data in urejanje poslovnih objektov

Kot je že bilo omenjeno, potrebujemo za hranjenje poslovnih objektov drugačno rešitev od tiste, ki jo ponuja modul Website. V ta namen uporabimo modul Data. Slednji podatke shranjuje v svoje delovno okolje (*workspace*), do njih pa imamo vseeno dostop iz predlog.

Pospopek za uvedbo nekega podatkovnega tipa zajema naslednje korake:

1. Ustvarimo **nov podatkovni tip**. Določimo mu ime (npr. „Kupon“), lokacijo v repozitoriju (npr. „/trgovina/akcije/kuponi“), kjer se bodo shranjevali podatki zanj (instance tega tipa).
2. Ustvarimo **dialog za urejanje** tega tipa, s čimer implicitno določimo tudi njegove lastnosti, ki jih bomo za posamezne instance tega tipa urejali. Če torej v dialog dodamo vnosno polje za datum in ga poimenujemo „Datum veljavnosti“, bodo imeli naši kuponi lastnost „Datum veljavnosti“, ki bo hranila nek datum.
3. Ustvarimo Javin razred (t.i. **model**), ki vsebuje metode, ki vračajo tiste vrednosti, ki jih bomo potrebovali v predlogi. Te metode sprogramiramo tako, da pridobivajo podatke iz JCR-ja. Kje v JCR-ju so podatki shranjeni pa vemo, ker smo sami definirali lokacijo shranjevanja v repozitoriju in poznamo strukturo podatkovnega tipa.
4. Sedaj lahko podatkovni tip uporabimo. Če želimo njegove lastnosti prikazati na kaki spletni strani, moramo definirati **predlogo** in njeno skripto. V definiciji predloge tako kot pri običajnih vsebinah nastavimo želeni dialog za urejanje (v tem primeru tistega, ki smo ga naredili v koraku 2) in skripto predloge, dodatno pa moramo še nastaviti v prejšnjem koraku omenjeni model, katerega metode bodo oskrbovale predlogo s podatki.

Uvedba nekega lastnega podatkovnega tipa oziroma urejanje poljubnih poslovnih objektov je, kot vidimo, bolj boleče kot v Parsek CMS 2.0. Vse podatkovne tipe moramo vnaprej predvideti in sprogramirati ustrezno logiko (model) ter vnesti ustrezne definicije v CMS.

5.2.5 Primerjava s Parsek CMS 2.0

Opazimo, da imata kljub bistvenim razlikam oba sistema eno skupno točko (poleg programskega jezika Java) in sicer hierarhično urejenost vsebin. Slednji princip je nasploh prisoten v skoraj vseh sistemih CMS in je glede na razširjenost očitno dobra praksa, ki pripomore k organiziranosti in preglednosti.

V osnovi je razlika med obema CMS-jema v namembnosti, saj je Magnolia, kot večina sistemov CMS, predstavitevno naravnana in v prvi vrsti namenjena izdelavi spletnih mest ter portalov (ne samo urejanju njihovih vsebin!), ne pa toliko poslovnih aplikacij oziroma kompleksnejših sistemov, ki zahtevajo urejanje vrste poslovnih objektov, čemur je primarno namenjen Parsek CMS 2.0.

Poslovne objekte moramo poznati vnaprej in ustrezno pripraviti Magnolio, da jih bo znala urejati ter prikazovati. Parsek CMS 2.0 je to v večji meri sposoben storiti samodejno in pred namestitvijo ni potrebna kostumizacija saj se prilagodi ciljni aplikaciji, med tem ko Magnolija s predlogami na nek način dejansko je ciljna aplikacija z integriranim CMS-jem.

Tako kot drugi CMS-ji s podobno namembnostjo, tudi Magnolija ponuja svoj sistem predlog. Slednje v Parsek CMS 2.0 ni na voljo in je prepuščeno razvijalcem ciljne aplikacije oziroma

ogrodju, ki se za razvoj tam uporablja. Parsek CMS 2.0 je zgolj sistem za upravljanje.

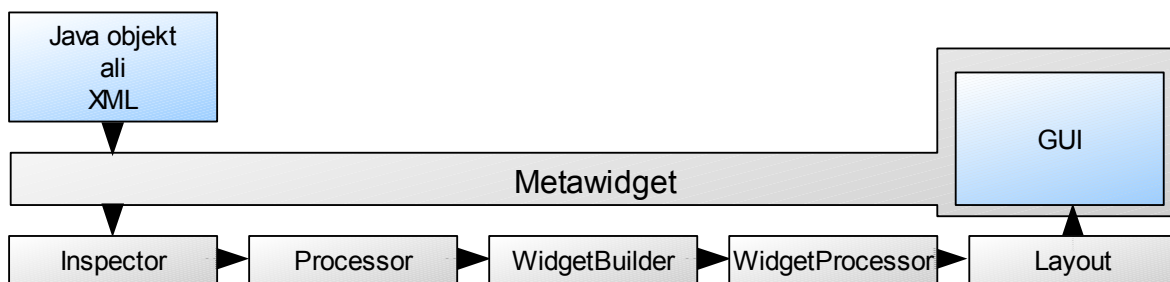
Ker za persistenco vsebin v Magnoliji skrbi JCR, je migracija obstoječih podatkov iz relacijskih podatkovnih baz sila nevhvaležno opravilo, ki vključuje razvoj namenske aplikacije, ki bo podatke brala iz relacijske podatkovne baze, jih ustrezno transformirala in shranila v JCR. Iz tega vidika je Magnolija popolnoma neprimerna za uporabo na obstoječih projektih in jo je smiselno uporabiti le za nove rešitve. Tudi integracija z drugimi sistemi se naprimer ne more izvajati preko skupne podatkovne baze temveč je potrebno izdelati namenski vmesnik, ki bi skrbel za transformacijo podatkov na podoben način kot pri migraciji.

Če potegnemo črto ugotovimo, da je Magnolia CMS primerna rešitev za nove projekte, ki se osredotočajo na spletno predstavitev (podjetja, posameznika, itd..) oziroma za raznorazne portale. Obstoječih projektov pa ne kaže migrirati, prav tako pa na Magnoliji ne kaže graditi novih projektov, ki so bolj sistemske narave. Slednjim je Parsek CMS 2.0 bolj pisan na kožo.

5.3 Metawidget

Metawidget sam po sebi ni sistem CMS, temveč zgolj rešitev, ki bi jo lahko uporabil pri implementaciji. Gre za orodje, ki sproti zgenerira vnosno formo za podan poslovni objekt. Počne torej podobno kot Parsek CMS 2.0 z analizo entitenih razredov in generiranjem generičnega urejevalnika.

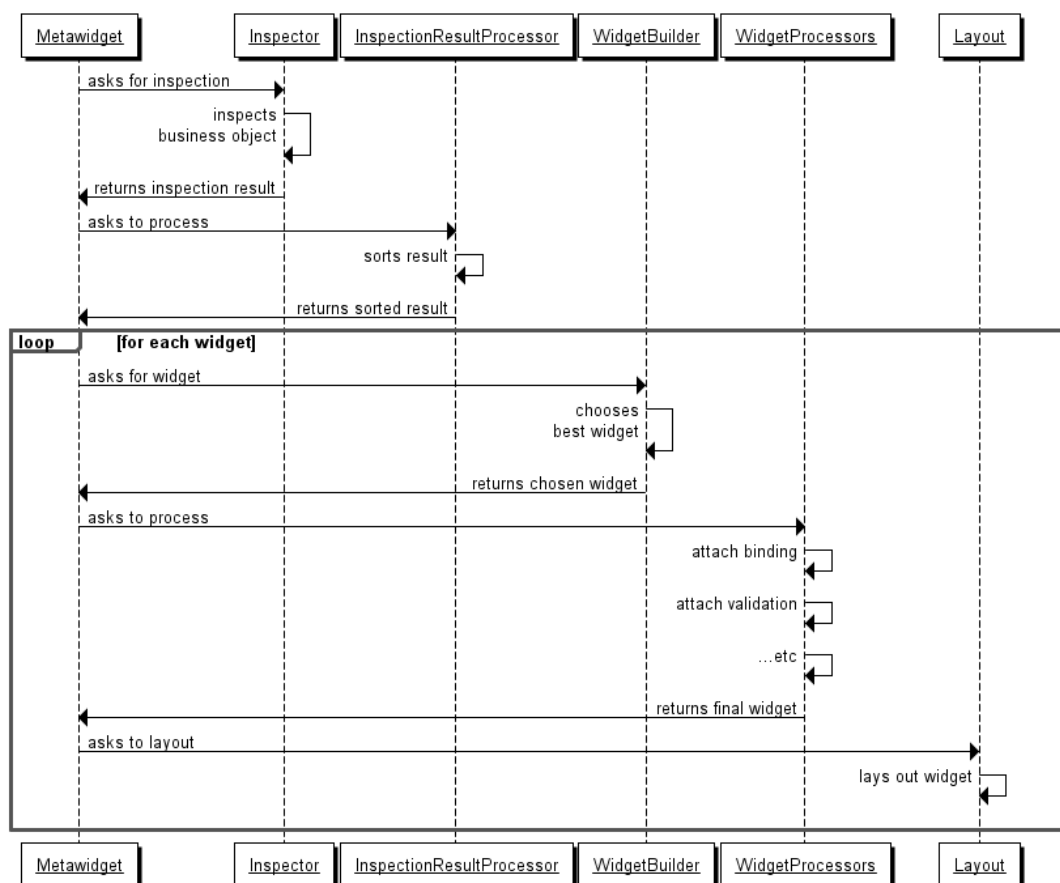
Analiza poslovnega objekta (podprti so Java objekti in XML datoteke), procesiranje rezultatov analize in generiranje uporabniškega spominja na cevovod z več koraki, pri čimer se za vsak korak uporabi primeren modul, ki implementira funkcionalnost koraka. Modularna zgradba omogoča enostavno dopolnjevanje Metawidgeta z novimi funkcionalnostmi in predelave, saj je projekt odprto-koden.



Slika 18: Cevovod

Metawidget prejme poslovni objekt in ga posreduje svojemu *Inspector*-ju, ki ga preuči (naprimer tipe lastnosti, če gre za Java objekt), *Processor* po potrebi dodatno obdelata rezultate analize (jih naprimer sortira, itd...), *WidgetBuilder* izbere in pripravi najprimernejšo vrsto vnosnega polja, *WidgetProcessor* pa to vnosno polje nadgradi z logiko za validacijo vnosa in poveže vsebino vnosnega polja z lastnostjo poslovnega objekta. Nazadnje *Layout* umesti vnosno polje na uporabniški vmesnik. Omenjeni moduli so zgolj abstraktni razredi, resnično delo pa opravljajo razredi, ki jih razširjajo. Obstaja naprimer *BaseObjectInspector*, ki zna analizirati Javine razrede in njegova razširitev *JpaInspector*, ki zna dodatno analizirati tudi JPA anotacije. Ogradje *Metawidget* podpira kar nekaj različnih „metawidgetov“,

„inspectorjev“ in „widget builderjev“, po potrebi pa lahko razvijalec implementira svoje module. Kaj določen tip modula v cevovodu počne, je bolj točno razvidno še iz naslednjega diagrama zaporedja:

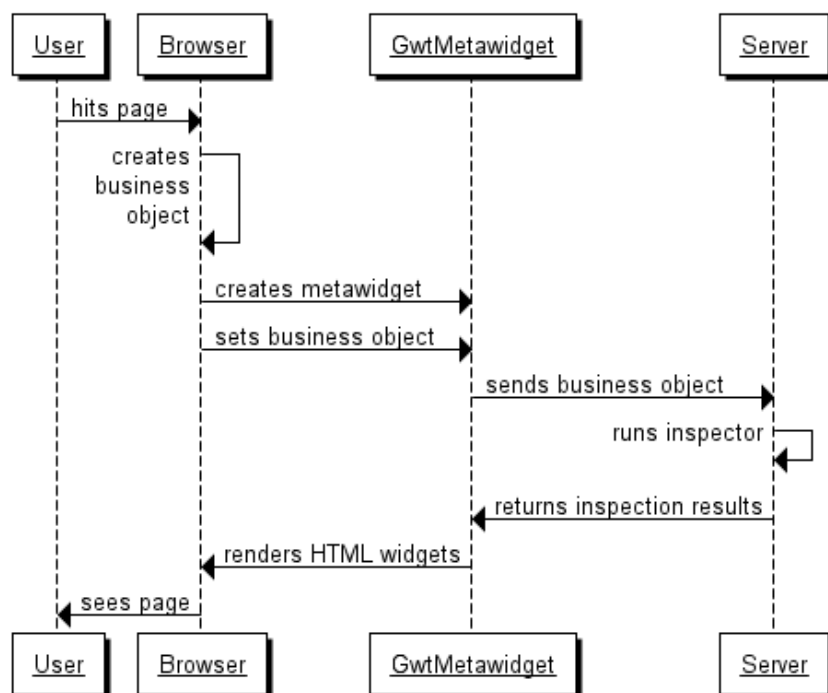


Slika 19: Od analize poslovnega objekta do uporabniškega vmesnika

Ogrodje Metawidget sicer podpira vrsto različnih implementacij modula *Metawidget*, od katerih je vsaka namenjena generiranju uporabniškega vmesnika v določeni tehnologiji. Tako so recimo na voljo *SwingMetawidget*, *HtmlMetawidget*, *GwtMetawidget*, itd... Še posebej zanimiv je slednji, saj je bilo že omenjeno, da je ravno GWT ena izmed v podjetju priporočenih tehnologij za izdelavo administrativnih orodij.

GwtMetawidget

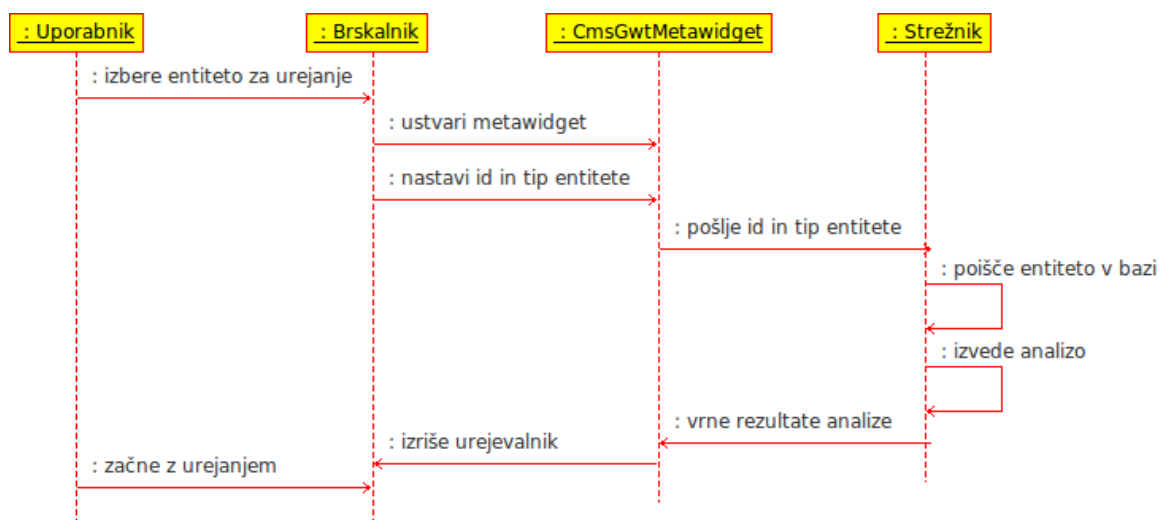
GwtMetawidget je, kot že samo ime pove, namenjen generiranju uporabniškega vmesnika za ogrodje GWT. Za razliko od večine ostalih, *GwtMetawidget* večino procesiranja preko RPC metode delegira strežniku, podobno kot to počne Parsek CMS 2.0. Pri natančnem pregledu delovanja in načina uporabe, ki sta razvidna iz diagrama zaporedja na naslednji strani pa že naletimo na prvo (od dveh pglavitnih) pomanjkljivost, zaradi katere je ogrodje Metawidget v nespremenjeni obliki neprimerno za uporabo v razvoju našega sistema CMS.



Slika 20: Diagram zaporedja pri uporabi GwtMetawidget-a

Na zgornjem diagramu zaporedja v oči zbode akcija „creates business object“, ki je na strani odjemalca. To ni funkcionalnost, ki jo iščemo, saj so naši poslovni objekti predstavljeni kot entitete oziroma instance entitetnih razredov in prihajajo iz strežnika. Privzeta implementacija *GwtMetawidget*-a torej predvideva, da se poslovni objekt kreira kar na odjemalcu in nato pošlje v analizo oziroma procesiranje na strežnik.

Na srečo je projekt odprto-koden, tako da bi bilo smiselno preučiti sledečo modifikacijo:



Slika 21: Modificiran diagram zaporedja

Sprogramirati je treba *CmsGwtMetawidget*, ki namesto celega poslovnega objekta prejme samo vrednost primarnega ključa in ime entitetnega tipa entitete, ki jo uporabnik želi urejati. Slednji podatki se pošljejo na strežnik, kjer moramo implementirati še svoj *CmsJpaInspector* (ni posebej označen na diagramu), ki namesto takojšnje analize poslovnega objekta najprej v bazi poišče ciljno entiteto in slednjo nato analizira ter preda naprej v „cevvod“. Rezultati analize se vrnejo na odjemalca, ki z obstoječim *GwtWidgetBuilder*-jem (ni posebej označen na diagramu, je pa del distribucije ogrodja Metawidget) ustvari ustrezen urejevalnik.

Dodatne modifikacije pa so pomembne še za drugo težavo in sicer je podpora JPA entitetam nekoliko pomankljiva. Metawidget namreč ne podpira vnosnih polj za zbirke (*Collections*), kot so sezname, množice, razpršene tabele, itd... To pomeni, da ne podpira urejanja povezav med entitetami tipa *many-to-many* in *one-to-many*. Zaradi tega je treba dopolniti tudi *GwtWidgetBuilder* in sicer implementirati generiranje vnosnih polj za urejanje zbirk. Modificirati je potrebno tudi *JpaInspector*, saj trenutno ne pozna *@ManyToMany* in *@OneToMany* anotacij.

Primerjava

Metawidget se v primeru generiranja uporabniškega vmesnika za GWT poslužuje skoraj identičnih konceptov kot Parsek CMS 2.0, le da ima slednji popolnejšo podporo za JPA (oziroma Hibernate) tako pri sami analizi kot tudi pri vnosnih poljih, ki so na voljo.

Videli smo, da bi za uporabo Metawidgeta v Parsek CMS 2.0 bilo potrebno kar nekaj modifikacij oziroma dopolnitev. Slednje pa je možno implementirati šele po predhodnim podrobnim spoznavanjem razvijalca z Metawidgetom in njegovo izvorno kodo. Čas, ki bi ga porabil za to, sem raje porabil za razvoj lastnega urejevalnika in analizatorja razredov. Lastna implementacija pa prinaša še druge prednosti in sicer dobro poznavanje celotnega sistema (lahko bi rekli „iz prve roke“) in nasploh proste roke pri načrtovanju arhitekture, brez potrebe po prilagajanju obstoječemu sistemu, ki arhitekturno ni optimalno zastavljen za rešitev problema.

Naj izpostavim še eno razliko, ki iz do sedaj povedanega morda ni tako očitna. Parsek CMS 2.0 entitete razrede analizira in rezultate analize shrani na seznam, ob nalaganju neke entitete pa mora tako obdelati le vrednosti njenih atributov, ne pa tudi entitetnega razreda, saj meta podatke o slednjem pridobi iz seznama. Metawidget pa vedno analizira podani objekt in vsakič znova opravi celotno analizo.

6. Zaključne ugotovitve

Rezultat dela, Parsek CMS 2.0, je resnično postal zanimivo in uporabno orodje za upravljanje s poslovnimi objekti, ki se kljub svoji „mladosti“ že uspešno uporablja v produkciji na nekaj projektih in ustrezno dosega naša pričakovanja. Končna rešitev zadovoljuje vse postavljene zahteve.

Še vedno pa nekoliko slabo skriva dejstvo, da je bil na začetku v uporabi za eno od naših spletnih trgovin, ki je bila pravzaprav poskusni zajček zanj. Mnogo funkcionalnosti in prilagoditev je bilo narejenih ravno na podlagi odzivov skrbnikov dotične spletne trgovine, zato je morda čutiti tudi rahlo pomanjkanje podpore za urejanje bolj portalskih strani oziroma strani z bogato vsebino. Tako bi lahko naprimer dodali še poseben urejevalnik za pisanje člankov, ki bi omogočal boljši pregled in lažje urejanje ter povezovanje vsebin v članek, itd... Po drugi strani pa se s tem spet oddaljujemo od zahteve po preprostosti in splošnosti.

Nastali sistem je tako s svojo napredno logiko za prebiranje podatkovnega modela in dinamično generiranje urejevalnika, namenjen bolj sistemskim projektom, pri katerih je v ospredju urejanje različnih poslovnih objektov, sama predstavitev (naprimer v obliki spletne strani) pa je bolj drugotnega pomena oziroma je zanj poskrbljeno v obliki raznih predlog, ki jih v celoti implementira ciljna aplikacija s poljubno tehnologijo. Če si kot primer spet pogledamo spletno trgovino, želimo predvsem urejati lastnosti nekega izdelka, le-te pa bodo za vse izdelke na spletni strani prikazane na isti način oziroma po neki vnaprej pripravljeni predlogi. Tukaj torej ne bomo potrebovali naprednega HTML urejevalnika, razen morebiti za podrobnejši opis izdelka, pa tudi pri tem ne bo kompliciranja.

Nove funkcionalnosti in izboljšave so že v načrtu in v razvoju, modurna zgradba, ki jo omogoča GWT, pa obljublja veliko možnosti za nadgradnje. Prioritetno bodo odpravljene najbolj očitne pomankljivosti – izboljšana validacija uporabnikovega vnosa, podpora pravicam in vlogam uporabnikov, verzioniranje vsebin (z vsakim urejanjem se inkrementira verzija neke vsebine in se hrani neko število prejšnjih verzij, na katere se lahko povrne - kot to naprimer že omogoča Magnolia), itd... Razvoj sistema sicer napreduje skladno s potrebami, ki jih sproti med uporabo identificirajo bodisi uporabniki sistema bodisi načrtovalci novih projektov.

Priloga A: Entiteta Message

Za večino atributov so uporabljene privzete vrednosti, po potrebi pa se lahko mnogo stvari dodatno specifikira oziroma pozovi privzete nastavitve z anotacijami (kot na primer dolžina niza pri atributu „subject“ in njegovo ime stolpca ter ime tabele).

```
@Entity //JPA anotacija, ki pomeni, da je ta razred entitetni razred
@Table(name = "simple_message") //ime tabele bo „simple_message“
public class Message {
    private String id;
    private String subject;
    private User sender;
    private List<User> recipients;
    private Date dateSent;
    private String content;

    @Id //označuje, da ta atribut predstavlja primarni identifikator
    @GeneratedValue(generator = "system-uuid") //ta vrednost naj bo samodejno generirana ob
    public String getId() { //vstavljanju v bazo
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    @Column(name = "message_subject", length = 100) //omejimo dolžino niza na 100 znakov
    public String getSubject() { //(privzeto sicer 255) in podamo ime
        return subject; //stolpca „message_subject“
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    @ManyToOne //JPA implementacija bo samodejno poskrbela za tuj ključ
    public User getSender() {
        return sender;
    }

    public void setSender(User sender) {
        this.sender = sender;
    }

    @ManyToMany //JPA implementacija bo samodejno poskrbela za povezovalno tabelo
    public List<User> getRecipients() {
        return recipients;
    }

    public void setRecipients(List<User> recipients) {
        this.recipients = recipients;
    }

    public Date getDateSent() {
        return dateSent;
    }
}
```

```
}  
  
public void setDateSent(Date dateSent) {  
    this.dateSent = dateSent;  
}  
  
@Lob //v bazo bo shranjen kot large object (odvisno od SUPB – npr. CLOB, TEXT, itd...)  
public String getContent() {  
    return content;  
}  
  
public void setContent(String content) {  
    this.content = content;  
}  
}
```

Priloga B: Meta opisi in Asset

1. *MetaAsset*

Lastnost	Pomen	Primer
String name	Ime, ki se uporablja v interne namene in enolično določa nek tip. Ponavadi je ime enako imenu entitetnega razreda. Če poznamo ime, lahko naprimer pridobimo njegovo definicijo: public MetaAsset getAsset(String name)	„Message“
String caption	Predstavlja uporabniku prijazno ime. Privzeto je kar enako zgoraj omenjenemu imenu, lahko pa bi določili svojo vrednost (naprimer „Sporočilo“ namesto „Message“).	„Message“
List<MetaField> fields	Seznam atributov oziroma polj, ki jih opisani entitetni tip vsebuje. Ta seznam vključuje tudi titleField in idField.	(našteta so imena polj) "id", "sender", "recipients", "dateSent", "subject", "content"
MetaField titleField	Določa polje, ki se uporablja za naslov oziroma napis vsebine. Predstavlja nek kratek povzetek vsebine in se izpisuje v tabelah, seznamih, itd... Za titleField se izbere tisti atribut, ki je anotiran z @TitleField, če pa takega ni, CMS sam izbere enega.	(podano je ime polja) „subject“
MetaField idField	Določa kateri atribut je primarni identifikator.	(podano je ime polja) „id“
List<MetaField> searchableFields	Seznam atributov, po katerih je dovoljeno iskanje z naprednim iskalnikom.	(našteta so imena polj) "id", "sender", "recipients", "dateSent", "subject", "content"
transient Class persistentClass	Entitetni razred, ki se v ogrodju Hibernate uporablja za persistenco	Message.class

	opisanega entitetnega tipa.	
boolean searchable	Ali naj iskalnik sploh išče po entitetah tega tipa.	true

2. MetaField

Lastnost	Pomen	Primer
String name	Ime polja, ki je enako imenu atributa, ki ga ta opis opisuje.	„sender“
FieldType type	Kakšnega tipa je atribut. Vpliva na tip vnosnega polja, ki bo zgenerirano. Možne vrednosti so opisane v poglavju o odjemalcu.	FieldType.SUB_ASSET
InputType inputType	Če je za nek tip možnih več različnih vnosnih polj, se generator urejevalnika pri izbiri opre še na to vrednost.	null
String label	Napis, ki bo viden pred vnosnim poljem. Če ni določen, se uporabi kar vrednost lastnosti name.	„sender“
boolean readOnly	Ali je vrednost samo za branje in je ni mogoče urejati.	false
boolean required	Ali mora vrednost nujno biti vnešena.	false
boolean subAsset	V kombinaciji z lastnostjo type v nekaterih primerih določa, ali to polje na nek način predstavlja povezavo z drugimi entitetami. FieldType.LIST in subAsset == true namprimer pomeni, da gre za <i>one-to-many</i> seznam povezanih entitet.	true
String subType	Ko je subAsset == true, ta lastnost določa, katerega tipa je ciljna entiteta. Uporabljeno je ime ustreznega MetaAsseta	„User“
boolean searchable	Določa, če je to polje vključeno v napredno iskanje. Polja iz seznama searchableFields imajo to vrednost nastavljeno na true.	true

<code>boolean unique</code>	Ali mora imeti to polje unikatno vrednost. Trenutno se sicer ne uporablja, je pa mišljeno za izboljšanje validacije uporabnikovega vnosa v prihodnjih verzijah.	<code>false</code>
<code>int order</code>	Vrstni red pri generiranju uporabniškega vmesnika. Privzeta vrednost je 100.	100
<code>Map<Object,String> possibleValues</code>	Seznam možnih vrednosti v primeru, da gre za <code>FieldType.ENUM</code> .	<code>null</code>
<code>transient Class fieldClass</code>	Katerega razreda je bila opisana lastnost entitete.	<code>User.class</code>

3. MapMetaField

`MapMetaField` je razširitev razreda `MetaField`. Uporabi se ga za opis entitetnih atributov, ki so razreda `Map`, npr. `Map<String, Article>`. Definira štiri dodatne lastnosti, ki opisujejo katerega tipa (oziroma razreda) je ključ in katerega tipa je vrednost.

Lastnost	Pomen
<code>FieldType keyType</code>	Kakšnega tipa so ključi. Vpliva na tip urejevalnika celice v tabeli (pojasnjeno v prilogi o odjemalcu).
<code>FieldType valueType</code>	Kakšnega tipa so vrednosti. Vpliva na tip urejevalnika celice v tabeli (pojasnjeno v prilogi o odjemalcu).
<code>String keyTypeSubType</code>	Ime razreda ključev, npr. „String“.
<code>String valueTypeSubType</code>	Ime razreda vrednosti, npr. „Article“.

4. Asset (instanca vsebine)

V objekte razreda `Asset` se shranijo vrednosti posameznih atributov neke entitete, ki jo naložimo za urejanje. Uporablja se jih pri komunikaciji med strežnikom in odjemalcem ter nasploh pri procesiranju na odjemalčevi strani. `Asset` impementira vmesnik `ModelData` iz ogrodja `GXT`, ki je definiran na sledeč način:

```

public static interface ModelData {
    <X> X get(java.lang.String s);

    java.util.Map<java.lang.String,java.lang.Object> getProperties();

    java.util.Collection<java.lang.String> getPropertyNames();

    <X> X remove(java.lang.String s);

    <X> X set(java.lang.String s, X x);
}

```

Vmesnik ModelData

Kot vidimo vsebuje tablo ključev in vrednosti. Za ključ uporabljeno ime nekega polja (lastnost name razreda MetaField), naprimer „title“, pripadajoča vrednost pa je vrednost, ki jo želimo pripisati temu polju, naprimer „Kontaktni podatki“. Ima pa Asset še nekaj drugih lastnosti, ki so opisane v spodnji tabeli. Opazimo, da ne vsebuje celotnega meta opisa (MetaAsset), temveč samo meta opise polj (seznam MetaField objektov).

Lastnost	Pomen
Serializable id	Vrednost primarnega ključa – ni nujno, da je tipa String.
String type	Ime ustrežujočega meta opisa (name iz razreda MetaAsset)
String title	Naslov oziroma povzetek. Uporablja se za prikaz v raznih seznamih oziroma preglednicah. To je hitrejše, kot pridobivanje meta opisa in ugotavljanja katero polje je uporabljeno za naslov.
List<MetaField> fields	Seznam polj. Le-ta je isti kot seznam polj iz ustreznega MetaAsset-a (tistega, katerega ime je enako tukajšnji lastnosti type)
boolean isNew	Označuje, ali gre za entiteto, ki je že persistirana v podatkovni bazi, ali za novo.

Za nekatere entitete, ki bi jih sicer lahko predstavili kot Asset, se iz praktičnih razlogov (iz vidika programiranja) uporabljajo namenski razredi. V veliki meri velja kar pravilo, da se Asset uporablja v urejevalnikih in pri rezultatih iskanja, spodnji razredi pa pri implementaciji večine ostalih funkcionalnosti, kjer ne gre za urejanje, kot so naprimer drevo kategorij, iskalnik oziroma izbirnik slik, itd...

Entitetni razred	Za uporabo v odjemalcu pretvorjen v:
Category	Folder
Image	Picture
com.parsek.cms.Relation	com.parsek.cmc.gxt.client.mvc.model.Relation

Folder vsebuje podatke o kategoriji, kot so njen primarni ključ, položaj znotraj neke kategorije (za urejanje vrstnega reda podkategorij), starševsko kategorijo, naslov (`title`), itd...

Picture poleg primarnega ključa hrani dimenzijo slike, ime datoteke, velikost datoteke, naslov slike (`title`) in datum zadnje spremembe. Sami binarni podatki datoteke pa tukaj niso shranjeni ampak se jih zaradi velikosti raje pridobiva in shranjuje z ločenim klicem na strežnik, kadar je to potrebno.

Relation, namenjen uporabi v odjemalcu, podobno kot ustrezen entitetni razred, vsebuje vrednost primarnega ključa, naslov, primarni ključ ciljne vsebine, tip ciljne vsebine in indeks za urejanje vrstnega reda.

Priloga C: MetaDataFactory algoritmi

1. createMetaAsset

```

public static MetaAsset createMetaAsset(final ClassMetadata cmd) throws Exception {
    MetaAsset ma = new MetaAsset();

    //preberemo entitetni razred, ki ga opisujejo meta podatki iz argumenta cmd
    Class clazz = cmd.getMappedClass(EntityMode.POJO);

    //ime tipa vsebine je kar kratko ime entitetnega razreda
    ma.setName(clazz.getSimpleName());

    //preberemo morebitno CMS anotacijo, ki vsebuje dodatno konfiguracijo tipa vsebine
    CMSAsset cmsasset = (CMSAsset) clazz.getAnnotation(CMSAsset.class);

    //če je z anotacijo določen napis polja ga uporabimo, sicer uporabimo kar ime
    String caption = clazz.getSimpleName();
    if (cmsasset != null) {
        caption = cmsasset.label();
    }
    ma.setCaption(caption);

    //na meta opis shranimo ime entitetnega razreda
    ma.setPersistentClass(clazz);

    TreeSet<MetaField> fields = new TreeSet<MetaField>(new FieldComparator());
    BeanInfo bi = Introspector.getBeanInfo(clazz);

    //seznam imen vseh lastnosti, ki jih Hibernate smatra za persistenčne
    ArrayList<String> list = new ArrayList<String>(Arrays.asList(cmd.getPropertyNames()));

    //seznam vseh lastnosti entitetnega razreda (tudi tranzitivne, ki niso persistenčne)
    Map<String, Field> beanFields = getAllFields(clazz);

    //iteriramo čez vse lastnosti entitetnega razreda
    for (PropertyDescriptor pd : bi.getPropertyDescriptors()) {
        //ali je trenutna lastnost primarni identifikator (to je poseben primer,
        //ker je ni na seznamu persistenčnih lastnosti in moramo biti pozorni na to)
        boolean id = pd.getName().equals(cmd.getIdentifierPropertyName());

        //če trenutna lastnost ni persistenčna temveč tranzitivna,
        //je prisotna na samem razredu, vendar je ni na seznamu persistenčnih lastnosti
        if (!list.contains(pd.getName()) && !id) continue;

        Field f = beanFields.get(pd.getName());
        Type fieldType = f == null ? pd.getPropertyType() : f.getGenericType();

        //za vsak slučaj še enkrat preverimo, da lastnost ni tranzitivna
        if (pd.getReadMethod() != null
            && pd.getReadMethod().getAnnotation(Transient.class) != null) {
            continue;
        }
    }
}

```

```

        //s posebno metodo zgeneriramo opis lastnosti in ga dodamo na seznam
        MetaField mf = MetaDataFactory.createMetaField(pd, fieldType, ma, id);
        if (mf != null) {
            fields.add(mf);
        }
    }

    //Če entiteta nima nobenega atributa, nimamo več kaj početi. To se načeloma
    //sicer ne zgodi.
    if (fields.isEmpty()) {
        return null;
    }

    /*
    Dodamo vsako polje na MetaAsset - MetaAsset.addField metoda je implementirana kot
    public void addField(MetaField field) {
        fields.add(field);
        if (field.isSearchable()) {
            searchableFields.add(field);
        }
    }
    */
    for (MetaField mf : fields) {
        ma.addField(mf);
    }

    //če noben atribut ni bil anotiran s @CMSTitle, kot naslov nastavimo kar prvo polje
    if (ma.getTitleField() == null) {
        ma.setTitleField(ma.getFields().get(0));
    }

    //nastavimo polje, ki je primarni identifikator
    MetaField id = ma.getField(cmd.getIdentifierPropertyName());
    if (id == null) {
        id = new MetaField(
            cmd.getIdentifierPropertyName(),
            FieldType.STRING,
            InputType.INPUT_TYPE_TEXTBOX,
            cmd.getIdentifierPropertyName(),
            true, false, false, null, true, 1000);
    }
    ma.setIdField(id);

    return ma;
}

```

2. createMetaField

```

public static MetaField createMetaField(PropertyDescriptor pd, Type fieldType,
                                         MetaAsset owner, boolean id) {
    String name;
    FieldType type;
    InputType inputType = null;
    String label = "";
    boolean readOnly = false;
    boolean required = false;

```

```

boolean subAsset = false;
boolean unique = false;
String subType1 = null;
String subType2 = null;
boolean searchable = false;

label = pd.getDisplayName();
name = pd.getDisplayName();
ParameterizedType parameterizedType = null;
Class returnType;
if (fieldType instanceof ParameterizedType) {
    parameterizedType = (ParameterizedType) fieldType;
    returnType = (Class) parameterizedType.getRawType();
} else {
    returnType = (Class) fieldType;
}

int order = 100;

//poiščemo getter za to lastnost
Method method = pd.getReadMethod();

CMSField cmsField = null;
Column column = null;

if (method != null) {
    //poiščemo CMS-jevo anotacijo, ki podaja dodatno konfiguracijo atributa
    cmsField = method.getAnnotation(CMSField.class);

    //poiščemo Hibernatovo anotacijo, ki določa dodatne lastnosti atributa
    column = method.getAnnotation(Column.class);

    //povozimo privzete vrednosti s tistimi, ki so podane z anotacijami
    if (cmsField != null) {
        label = cmsField.label();
        order = cmsField.order();
        required = cmsField.required();
        readOnly = cmsField.readOnly();
        if (cmsField.inputType() != InputType.GUESS) {
            inputType = cmsField.inputType();
        }
        if (!cmsField.editable()) {
            return null;
        }
    }

    if (column != null) {
        unique = column.unique();
    }
}

//primarnega identifikatorja ne urejamo, razen če to omogočimo
//s CMS-jevo anotacijo.
if (id && cmsField == null) {
    return null;
}

//poskusimo določiti FieldType za enostavno polje.
type = getFieldType(returnType);

//primitivni javini tipi ne morejo imeti vrednosti null
if (!required && returnType.isPrimitive()) {
    required = true;
}

```

```

//poskus, da bi določili enostavno polje ni uspel, torej potrebujemo
//kompleksno polje, oziroma gre za neke vrste povezavo z drugo entiteto.
if (type == null) {
    ManyToOne mto = method.getAnnotation(ManyToOne.class);
    OneToMany otm = method.getAnnotation(OneToMany.class);
    ManyToMany mtm = method.getAnnotation(ManyToMany.class);
    OneToOne oto = method.getAnnotation(OneToOne.class);
    if (mto != null) {
        inputType = InputType.INPUT_TYPE_LOOKUP;
        type = FieldType.SUB_ASSET;
        subType1 = returnType.getSimpleName();
        subAsset = true;
    } else if (otm != null) {
        inputType = InputType.INPUT_TYPE_LIST;
        type = FieldType.LIST;
        if (parameterizedType == null) {
            return null;
        }
        Type paramType = parameterizedType.getActualTypeArguments()[0];
        subType1 = ((Class) paramType).getSimpleName();
        subAsset = true;
    } else if (mtm != null) {
        inputType = InputType.INPUT_TYPE_LIST;
        type = FieldType.LIST;
        if (parameterizedType == null) {
            return null;
        }
        Type paramType = parameterizedType.getActualTypeArguments()[0];
        subType1 = ((Class) paramType).getSimpleName();
        subAsset = true;
    } else if (oto != null) {
        inputType = InputType.INPUT_TYPE_LOOKUP;
        type = FieldType.SUB_ASSET;
        subType1 = returnType.getSimpleName();
        subAsset = true;
    } else if (List.class.isAssignableFrom(returnType)) {
        type = FieldType.LIST;
        inputType = InputType.INPUT_TYPE_LIST;
    } else {
        //nepredvidena situacija...
        log.warn("We have no idea what kind of field this is!");
        type = FieldType.STRING;
    }

    if (Map.class.isAssignableFrom(returnType)) {
        type = FieldType.MAP;
        inputType = InputType.INPUT_TYPE_MAP;
        subAsset = false;
    }
} else {
    //privzeto lahko napredno iskanje uporabljamo s katerikoli enostavnim atributom
    searchable = true;
}

if (type == FieldType.STRING) {
    Lob lob = method.getAnnotation(Lob.class);
    //če gre za CLOB, to običajno pomeni HTML.
    if (lob != null) {
        if (inputType == null) {
            inputType = InputType.INPUT_TYPE_HTML EDIT;
        }
        //ne dovolimo naprednega iskanja po CLOB poljih
        searchable = false;
    }
}

```

```

    } else if (column != null && column.length() > 260) {
        //če gre za običajen niz, vendar je ta zelo dolg, bomo
        //potrebovali večvrstično vnosno polje
        if (inputType == null) {
            inputType = InputType.INPUT_TYPE_TEXTAREA;
        }
    } else {
        searchable = true;
    }
}

//če ni šlo za poseben primer, kjer smo tip vnosnega polja že določili,
//ga določimo sedaj.
if (inputType == null) inputType = getInputType(returnType);

//preverimo če polje predstavlja sliko ali seznam slik, saj v tem primeru za
//vnos uporabimo poseben izbirnik za slike.
if ((type == FieldType.SUB_ASSET || type == FieldType.LIST)
    && (subType1 != null && subType1.equals(Image.class.getSimpleName()))) {
    inputType = InputType.INPUT_TYPE_IMAGE_PICKER;
}

MetaField ret = null;

//pri atributih tipa Map je potrebno določiti še nekatere
//lastnosti ključev in vrednosti.
if (type == FieldType.MAP) {
    //katerega tipa sta ključ in vrednost
    Type paramType1 = parameterizedType.getActualTypeArguments()[0];
    subType1 = ((Class) paramType1).getSimpleName();
    Type paramType2 = parameterizedType.getActualTypeArguments()[1];
    subType2 = ((Class) paramType2).getSimpleName();

    MapKeyManyToMany mapKeyManyToMany = method.getAnnotation(MapKeyManyToMany.class);

    //MapMetaField je razširitev MetaField, ki vsebuje dodatne informacije
    //o ključih in vrednostih
    MapMetaField r = new MapMetaField(name, type, inputType, label, readOnly,
        required, subAsset, subType1, false, order);
    r.setKeyTypeSubType(subType1);
    r.setValueTypeSubType(subType2);
    FieldType keyType = getFieldType((Class) paramType1);
    if (keyType == null) {
        keyType = FieldType.SUB_ASSET;
    }
    r.setKeyType(keyType);

    FieldType valueType = getFieldType((Class) paramType2);
    if (valueType == null) {
        valueType = FieldType.SUB_ASSET;
    }
    r.setValueType(valueType);
    ret = r;
} else {
    ret = new MetaField(name, type, inputType, label, readOnly,
        required, subAsset, subType1, searchable, order);
    ret.setUnique(unique);
}

//V primeru enumeracije zgradimo seznam vseh možnih vrednosti
if (type == FieldType.ENUM) {
    Map<Object, String> possibilities = new LinkedHashMap<Object, String>();

    for (Object e : returnType.getEnumConstants()) {

```

```

        possibilities.put(((Enum) e).ordinal(), ((Enum) e).name());
    }
    ret.setPossibleValues(possibilities);
    ret.setFieldClass(returnType);
}

//če je polje anotirano s TitleField, to opredelimo tudi na MetaAssetu
if (method != null) {
    TitleField ft = method.getAnnotation(TitleField.class);

    if (ft != null) {
        ret.setSearchable(true);
        owner.setTitleField(ret);
    }
}
return ret;
}

```

3. Pomožne metode

```

private static FieldType getFieldType(Class type) {
    if (type.isEnum()) {
        return FieldType.ENUM;
    } else if (java.util.Date.class.isAssignableFrom(type)) {
        return FieldType.DATE;
    } else if (Integer.class.isAssignableFrom(type) || "int".equals(type.getName())) {
        return FieldType.INTEGER;
    } else if (Long.class.isAssignableFrom(type) || Long.class.isAssignableFrom(type)
        || "long".equals(type.getName())) {
        return FieldType.LONG;
    } else if (Float.class.isAssignableFrom(type) || Double.class.isAssignableFrom(type)
        || "double".equals(type.getName()) || "float".equals(type.getName())) {
        return FieldType.FLOAT;
    } else if (Boolean.class.isAssignableFrom(type) || "boolean".equals(type.getName())) {
        return FieldType.BOOLEAN;
    } else if (String.class.isAssignableFrom(type)) {
        return FieldType.STRING;
    } else if (byte[].class.isAssignableFrom(type) || Byte[].class.isAssignableFrom(type)) {
        return FieldType.BINARY;
    } else {
        return null;
    }
}

```

```

private static InputType getInputType(Class type) {
    if (type.isEnum()) {
        return InputType.INPUT_TYPE_LOOKUP;
    } else if (java.util.Date.class.isAssignableFrom(type)) {
        return InputType.INPUT_TYPE_DATE;
    } else if (Integer.class.isAssignableFrom(type) || "int".equals(type.getName())) {
        return InputType.INPUT_TYPE_TEXTBOX;
    } else if (Long.class.isAssignableFrom(type) || Long.class.isAssignableFrom(type)
        || "long".equals(type.getName())) {
        return InputType.INPUT_TYPE_TEXTBOX;
    } else if (Float.class.isAssignableFrom(type) || Double.class.isAssignableFrom(type)
        || "double".equals(type.getName()) || "float".equals(type.getName())) {
        return InputType.INPUT_TYPE_TEXTBOX;
    } else if (Boolean.class.isAssignableFrom(type) || "boolean".equals(type.getName())) {
        return InputType.INPUT_TYPE_TEXTBOX;
    } else if (String.class.isAssignableFrom(type)) {
        return InputType.INPUT_TYPE_TEXTBOX;
    }
}

```

```
        return InputType.INPUT_TYPE_TEXTBOX;
    } else if (byte[].class.isAssignableFrom(type) || Byte[].class.isAssignableFrom(type)) {
        return InputType.INPUT_TYPE_FILE_UPLOAD;
    } else {
        return null;
    }
}
```

Priloga D: pomembnejše metode razreda Core

```
/**
 * Pokliče se po uspešni prijavi uporabnika in ustvari komponento MetaAssets ter naloži morebitno
 * dodatno konfiguracijo iz konfiguracijske XML datoteke.
 */
void start()

/**
 * Vrne vse tipe vsebine, ki so na voljo. Ključ je MetaAsset.name, vrednost pa MetaAsset.caption
 */
Map<String, String> getAvailableAssetsMap();

/**
 * Vrne meta opise tistih tipov vsebine, po katerih je dovoljeno oziroma možno napredno iskanje.
 */
List<MetaAsset> getSearchableAssets();

/**
 * Vrne imena tipov vsebine, ki so na voljo (podobno kot getAvailableAssetsMap).
 */
List<String> getAvailableAssetsList();

/**
 * Za podano ime tipa vsebine, vrne njegov meta opis.
 */
MetaAsset getAssetDef(String assetType)

/**
 * Vrne kategorijo, ki je trenutno izbrana v drevesu kategorij, oziroma je njena vsebina
 * prikazana v preglednici na desni.
 */
String getWorkingCategoryID();

/**
 * Posodobi informacijo o tem, katera kategorija je trenutno izbrana v drevesu kategorij.
 */
void setWorkingCategory(final String path, final String workingCategoryID);

/**
 * Ustvari novo entiteto, ki ustreza tipu vsebine s podanim imenom.
 */
Object createNewEntity(String type);

/**
 * V bazi poišče in vrne (če obstaja, sicer vrne null) entiteto, ki ustreza podani vrednosti
 * primarnega ključa in podanemu tipu vsebine.
 */
Object getEntity(Object id, String type);

/**
 * Doda podano entiteto v podano kategorijo (ustvari novo povezavo – entiteto tipa Relation - in
```

```
jo shrani).
*/
void addAssetToCategory(Object entity, String categoryId);

/**
Kot prejšnja metoda, le da sprejme druge argumente.
*/
void addAssetToCategory(String assetId, String assetType, String categoryId);

/**
Posodobi naslov na vseh povezavah (entitetah tipa Relation), ki s kategorijami povezujejo
vsebino s podano vrednostjo primarnega identifikatorja in podanega tipa.
*/
void updateRelations(String assetId, String assetType, String title);

/**
Izbriše entiteto s podano vrednostjo primarnega identifikatorja in tipa.
*/
boolean deleteEntity(Serializable assetId, String assetType);

/**
Naredi in vrne kopijo entitete, ki ustreza argumentoma, če le-ta obstaja.
*/
Object cloneObject(Serializable assetId, String assetType);

/**
Podano entiteto pretvori v Asset.
*/
Asset getAsset(Object entity)
```

Priloga E: pretvorba entitete v Asset

```
public Asset getAsset(Object entity, boolean refresh) {
    if (entity == null) return null;

    String type = core.getAssetNameFromClass(entity.getClass());

    Asset result = new Asset();

    try {
        //entiteto ovijemo v BeanMap, ki izpostavi vse lastnosti objekta
        //kot pare ključ - vrednost, kjer je ključ ime lastnosti.
        BeanMap map = new BeanMap(entity);

        //pridobi ustrezen meta opis
        MetaAsset ma = core.getAssetDef(type);

        //pridobi vrednost primarnega ključa
        Serializable id = (Serializable) (
            map.get(ma.getIdField().getName()) == null
                ? null
                : map.get(ma.getIdField().getName()));

        if (refresh) {
            Class clazz = entity.getClass();
            if (clazz.getSimpleName().contains("$")) {
                clazz = clazz.getSuperclass();
            }
            entity = entityManager.getReference(clazz, id);
            map = new BeanMap(entity);
        }

        //nastavi nekatere osnovne podatke
        result.setId(id);
        result.setType(type);
        result.setNew(id == null);

        //iteriraj čez vsa polja
        for (MetaField mf : ma.getFields()) {
            String fieldName = mf.getName();

            //na podlagi imena pridobi vrednost lastnosti
            Object obj = map.get(fieldName);

            if (mf.isSubAsset() && mf.getType() != FieldType.LIST) {
                //polje predstavlja povezavo na drugo entiteto - z metodo
                //getEntityIdTitleMapping predelamo v
                //par primarnega ključa in naslova (title).
                if (mf.getType() == FieldType.SUB_ASSET) {
                    BaseModelData bm =
                        getEntityIdTitleMapping(obj, core.getAssetDef(mf.getSubType()));
                    result.set(fieldName, bm);
                }
            } else if (mf.getType() == FieldType.LIST) {
```

```

//seznam entitet predelamo v pare primarnih ključev in naslovov.
List<ModelData> subAssetList = new ArrayList<ModelData>();
if (mf.isSubAsset()) {
    List subAssets = (List) obj;
    if (subAssets != null) {
        for (Object objSubAsset : subAssets) {
            if (objSubAsset == null) {
                continue;
            }
            BaseModelData bm =
                getEntityIdTitleMapping(objSubAsset,
                    core.getAssetDef(mf.getSubType()));
            subAssetList.add(bm);
        }
    }
    result.set(fieldName, subAssetList);
} else {
    log.warn("List of elements is not supported!");
}
} else if (obj == null) {
    //če je vrednost null, se ne ubadamo več
    result.set(fieldName, null);
} else if (!(obj instanceof Serializable)) {
    //transientne lastnosti se sicer ne bi smele pojavljati...
    log.warn("Trying to serialize a non-serializable class "
        + obj.getClass().getSimpleName() + " for field "
        + fieldName + " on asset "
        + ma.getName() + "!");
    //za vsak slučaj shranimo vsaj besedilno predstavitev objekta.
    result.set(fieldName, obj.toString());
} else if (mf.getType() == FieldType.BINARY) {
    //poseben primer - podatki za tovrstna polja se nalagajo in shranjujejo
    //posebej z ločenimi klici na strežnik.
    result.set(fieldName, "");
} else if (mf.getType() == FieldType.ENUM) {
    BaseModelData data = new BaseModelData();
    data.set("title", ((Enum) obj).name());
    data.set("value", ((Enum) obj).ordinal());
    result.set(fieldName, data);
} else if (mf.getType() == FieldType.MAP) {
    MapMetaField mmf = (MapMetaField) mf;
    Map m = (Map) obj;

    //tovrstna polja pretvorimo v običajen seznam, ki kot elemente
    //vsebuje pare ključ - vrednost.
    List<ModelData> mapList = new ArrayList<ModelData>(m.size());
    for (Object key : m.keySet()) {
        ModelData mdEntry = new BaseModelData();

        if (mmf.getKeyType() == FieldType.SUB_ASSET) {
            //če gre za entiteto in ne za enostaven tip, jo naložimo
            //in shranimo trojček id - title - type, ki ga vrne
            //metoda wrapSubAsset
            BeanMap keyMap = new BeanMap(key);
            Asset subAsset = getAsset((Serializable) keyMap.get("id"),
                mmf.getKeyTypeSubType());
            mdEntry.set("key", wrapSubAsset(subAsset));
        } else {
            mdEntry.set("key", key);
        }

        if (mmf.getValueType() == FieldType.SUB_ASSET) {
            BeanMap valueMap = new BeanMap(m.get(key));
            Asset subAsset = getAsset((Serializable) valueMap.get("id"),

```

```

        mmf.getValueTypeSubType());
        mdEntry.set("value", wrapSubAsset(subAsset));
    } else {
        mdEntry.set("value", m.get(key));
    }

    mapList.add(mdEntry);
}
result.set(fieldName, mapList);
} else {
    //trenutna lastnost je očitno enostavnega tipa in
    //samo nastavimo njeno vrednost na asset.
    result.set(fieldName, obj);
}

result.getFields().add(mf);
}

//nastavimo še naslov
if (ma.getTitleField() != null) {
    Object t = map.get(ma.getTitleField().getName());
    result.setTitle(t != null ? t.toString() : "<no title>");
}
} catch (CMCException e) {
    log.error("could not get asset", e);
}

return result;
}

private ModelData wrapSubAsset(Asset asset) {
    ModelData md = new BaseModelData();
    md.set("id", asset.getId());
    md.set("title", asset.getTitle());
    md.set("type", asset.getType());

    return md;
}

private Object getTitleFieldValue(Object entity, MetaField mf) {
    try {
        BeanMap map = new BeanMap(entity);
        if (mf.isSubAsset()) {
            Object ent = map.get(mf.getName());
            MetaAsset ma = core.getAssetDef(mf.getSubType());
            MetaField tf = ma.getTitleField();
            return getTitleFieldValue(ent, tf);
        } else {
            Object toReturn = map.get(mf.getName());
            if (toReturn != null && toReturn instanceof Enum) {
                return toReturn.toString();
            }
            return toReturn;
        }
    } catch (CMCException e) {
        log.error("Could not get title field ", e);
        return "<could not resolve title>";
    }
}

```

Priloga F: Opisi storitev (services)

Storitve na strežniku so odjemalcu na voljo preko oddaljenih klicev metod (RPC – „*Remote Procedure Call*“). Posamezna storitev je tako razred, ki implementira množico metod, do katerih ima odjemalec dostop in jih lahko pokliče.

CmcService

Ta storitev trenutno omogoča prijavo in odjavo uporabnika. Namenjena je torej nekakšnim splošnim opravilom, trenutno pa še nima veliko funkcionalnosti, zato ji tukaj ne bomo posvetili kaj več pozornosti.

EditorService

Ta storitev nudi direktno podporo urejevalniku. Omogoča nalaganje in shranjevanje vsebin (objektov razreda `Asset`) in delov vsebin, ki se nalagajo dinamično, ko je urejevalnik že zgeneriran, nadalje omogočajo kreiranje in kloniranje obstoječih vsebin, itd... V spodnji tabeli so opisane oddaljene metode, ki so na voljo odjemalcu:

<code>Asset getAsset(Serializable id, String type)</code>
Na podlagi podane vrednosti primarnega ključa (<code>id</code>) in tipa (<code>type</code>) naloži entiteto iz baze, jo pretvori v <code>Asset</code> in vrne odjemalcu.
<code>List<ModelData> getComboData(String type)</code>
Uporablja se za polnjenje padajočih seznamov, v katerih uporabnik izbere ciljno entiteto pri <i>many-to-one</i> in <i>one-to-one</i> povezavah. Na podlagi podanega tipa, poišče vse entitete tega tipa in vrne seznam trojčkov <i>id-title-type</i> (nesmiselno je namreč odjemalcu poslati celoten <code>Asset</code> , ker za prikaz potrebuje samo omenjene vrednosti).
<code>Asset save(Asset editableAsset)</code>
Naloži entiteto iz baze (če že obstaja), ki ustreza podanemu <code>Asset</code> -u ter posodobi njeno stanje v bazi. Če entiteta še ne obstaja, ustvari novo.
<code>Map<String, String> getAvalibaleNewAssetList()</code>
Vrne imena in napise (vrednosti lastnosti <code>MetaAsset.name</code> in <code>MetaAsset.caption</code>) vseh tipov

vsebin, ki jih lahko ustvarjamo.
<code>ListLoadResult<ModelData> getSubassetsForField(String assetType, Serializable assetId, String fieldName)</code>
Polje z imenom, podanim v <code>fieldName</code> , predstavlja atribut entitete, ki vsebuje seznam povezanih entitet z <i>one-to-many</i> ali <i>many-to-many</i> povezavo. Metoda poišče povezane entitete in jih pretvori v seznam parov <i>id – title</i> (ponovno ni smiselno pošiljati celotnih Asset-ov). Urejevalnik se je poslužuje, da s podatki zapolni preglednico za urejanje takih povezav (več o tem v poglavju o odjemalcu). Kot primer lahko vzamemo <code>Message.recipients</code> , kjer bi <code>fieldName</code> imel vrednost „recipients“, rezultat pa bi bil seznam vseh prejemnikov.
<code>PagingLoadResult<ModelData> searchForAssets(String assetType, String query, PagingLoadConfig config)</code>
Vrne seznam entitet, kjer atribut, ki predstavlja naslov (<code>MetaAsset.titleField</code>), ustreza iskalni poizvedbi <code>query</code> (v smislu poizvedbe „... <i>where [ime_atributa] like %query% ...</i> “). Uporablja se v oknu za izbiro ciljne entitete, ki je opisano v poglavju o odjemalcu.
<code>ListLoadResult<Picture> getPhotos(String query)</code>
Podobno kot <code>searchForAssets</code> , le da je namenjena naloženim slikovnim datotekam in vrača specializiran razred <code>Picture</code> namesto generičnega <code>Asset</code> .
<code>Asset cloneAssetFromRelation(Relation original)</code>
Klonira vsebino, ki jo z neko kategorijo povezuje podana povezava, tako da izlušči potrebne argumente in pokliče drugo metodo, ki opravi dejansko delo: <pre>return cloneAsset(original.getAssetId(), original.getAssetType());</pre>
<code>Asset cloneAsset(Serializable assetId, String assetType)</code>
Klonira vsebino, ki ustreza podanim argumentom in vrne klon.
<code>Picture getPhoto(String id)</code>
Na podlagi podane vrednosti primarnega ključa poišče in vrne podatke o sliki.

GridService

Ko v drevesu kategorij v odjemalcu uporabnik izbere eno kategorijo, se v preglednici izpiše, katere vsebine so povezane z izbrano kategorijo. Metod te storitve se poslužuje ta preglednica. Ker je v neko kategorijo lahko uvrščenih mnogo vsebin, je pri nekaterih metodah uporabljena tudi podpora za paginacijo, ki jo zagotavlja ogrodje GXT.

Za lažje razumevanje:

PagingLoadResult je seznam objektov, ki ima poleg samega seznama še dodatne lastnosti, kot so številka trenutne strani, velikost strani in število vseh zadetkov, da odjemalec te

informacije lahko prikaže v orodni vrstici za paginacijo.

PagingLoadConfig je objekt, iz katerega strežnik lahko razbere koliko zadetkov naj vrne (velikost strani) in pri katerem zadetku naj začne (številka strani, iz česar lahko izračuna številko prvega zadetka s formulo $\text{številka_strani} * \text{velikost_strani}$).

<code>PagingLoadResult<Relation> getRelationsForCategory(String categoryId, PagingLoadConfig loadConfig)</code>
Vrne seznam povezav, ki s podano kategorijo povezujejo vsebine oziroma odgovori na vprašanje „Katere vsebine so povezane s to kategorijo?“.
<code>void saveRelationOrdering(List<Relation> relations, int page, int pageSize)</code>
Uporabnik lahko v preglednici z „drag and drop“ tehniko preureja vrstni red. Ko „izpusti“, kar je „zagrabil“, odjemalec pokliče to metodo in tako sporoči nov vrstni red na trenutni strani (vrstni red je razviden iz seznama relations).
<code>List<Relation> moveToNextPage(List<Relation> relations, int page, int pageSize)</code>
Ker je na eni strani prikazano samo določeno število elementov, je z drag and drop tehniko možno preurejati vrstni red samo znotraj ene strani. Če želi uporabnik izbrane elemente prenesti na naslednjo stran, mora pritisniti poseben gumb v orodni vrstici, odjemalec pa potem pokliče to metodo. Rezultat, ki ga ta funkcija vrača, je nov seznam elementov za trenutno prikazano stran (med katerimi je tudi toliko elementov iz naslednje strani, kot jih je bilo izpodrinjenih na račun tistih, ki so bili prestavljeni nanjo). Recimo da je na strani prikazanih 50 elementov, mi pa smo jih 5 prestavili na naslednjo stran. Prvih 5 elementov, ki so že bili na naslednji strani, se po novem prikazuje na koncu trenutne strani, tako da odjemalec kot rezultat prejme seznam prvih 45 elementov trenutne strani in prvih 5 elementov naslednje strani, ki po novem spadajo na konec trenutne strani.
<code>List<Relation> moveToPreviousPage(List<Relation> relations, int page, int pageSize)</code>
Kot prejšnja metoda, le da gre za premik na prejšnjo stran.
<code>List<Boolean> removeAssetsFromCategory(List<Relation> relations, String categoryId);</code>
Odstrani vsebine iz kategorije (v bazi izbriše podane povezave, ki povezujejo vsebine s kategorijo). Dodatni argument categoryId je tu samo zaradi preverjanja konsistentnosti, saj že sama povezava vsebuje id kategorije. Samih vsebin ne izbriše iz baze. Seznam, ki ga funkcija vrača nakazuje uspešnost odstranjevanja za posamezno povezavo.
<code>List<Boolean> deleteAssets(List<Relation> relations);</code>
Poleg brisanja povezav poskusi izbrisati tudi vsebino, če le ta ni povezana še s kako kategorijo, sicer operacije ne uspe (uspešnost brisanja vsake od podanih vsebin je zaznamovana v seznamu, ki ga funkcija vrača).

```
Boolean deleteAsset(Serializable assetId, String assetType);
```

Kot prejšnja funkcija, le da poskusi izbrisati posamezno vsebino in ne seznama ter kot argument ne vzame povezave temveč direktno primarni ključ vsebine.

ProductEditorService

Storitev je namenjena posebnemu urejevalniku, s katerim se ureja prodajne artikule v spletni trgovini (več o tem urejevalniku je v poglavju o odjemalcu). Ta urejevalnik se sicer še vedno v veliki meri poslužuje metod storitve EditorService, tukaj pa so implementirane še nekatere specializirane metode.

```
List<Property> getProperties(String tangibleId)
```

Vrne seznam lastnosti nekega izdelka. Lastnost (Property) v tem primeru pomeni naprimer velikost zaslona pri monitorju, količina pomnilnika pri računalniku, itd...

```
NewTangible saveTangible(Asset asset)
```

Shrani urejani izdelek. Vrnjeni objekt vsebuje nekaj informacij, ki jih odjemalec potrebuje po shranjevanju za posodobitev prikazanih podatkov.

```
List<Property> getProperties(String tangibleId, String groupId)
```

Pokliče se ob spremembi produktne skupine, v katero spada urejani izdelek in vrne samo lastnosti nove skupine. Če imata stara in nova skupina kaj istih lastnosti, se njihove vrednosti ohranijo (če nek izdelek prestavimo iz skupine zaslonov v skupino televizorjev, se naprimer ohrani velikost zaslona).

SearchService

SearchService dodaja podporo za iskanje. Na voljo sta dve vrsti iskanja. Eno je splošno iskanje z uporabo t.i. „full text search“, za kar se uporablja tehnologija Hibernate Search. Da ga lahko uporabljamo, moramo na entitetnih razredih dodati nekaj anotacij, s katerimi povemo katera polja naj bodo vključena v iskanje oziroma analizirana in indeksirana.

Ta funkcionalnost deluje v več korakih. Odjemalec najprej na strežnik sporoči iskalno poizvedbo, njeni rezultati pa se na strežniku začasno shranijo v okviru uporabnikove seje. Odjemalec nato z nadaljnjimi zahtevki pridobiva rezultate iskanja po straneh (paginacija).

Podporo tovrstnemu iskanju sem omenil zaradi popolnosti diplomske naloge oziroma celovitosti opisa CMS-ja, sicer pa sam osebno nisem implementator splošnega iskanja.

Druga vrsta iskanja je napredno iskanje, pri čimer gre za iskanje po izbranih lastnostih izbranega entitetnega tipa oziroma tipa vsebine. Uporabnik najprej izbere po katerem tipu bo iskal (npr. Article), nato pa izbere (obkljuka) še po katerih atributih tega tipa in za izbrane attribute določi iskalni pogoj. Pri atributih tipa String, kot iskalni pogoj vnese niz, ki se mora

pojavitvijo znotraj vrednosti. Če naprimer želi poiskati vse članke, ki v naslovu vsebujejo besedo „počitnice“, izbere tip „Article“ in njegov atribut „Title“ ter kot iskalni niz za ta atribut vnese „počitnice“. Pri številskih atributih je potrebno vnesti interval, prav tako pri atributih z datumom. Podprti so še boolean atributi. Ostali tipi, ki niso enostavni, zaenkrat še niso podprti, vendar v praksi do sedaj tudi še nismo občutili večje potrebe po podpori tovrstnih atributov v naprednem iskanju.

<code>List<MetaAsset> getSearchables();</code>
Vrne seznam tipov, po katerih je možno napredno iskanje. Odjemalec uporabniku ponudi iskanje po tipih iz tega seznama.
<code>PagingLoadResult<Asset> submitSearch(List<ModelData> query, MetaAsset asset, PagingLoadConfig config);</code>
Parameter query je seznam z iskalnimi poizvedbami za attribute, ki jih je uporabnik izbral. Izvede se poizvedba v podatkovni bazi, ki vrne ustrezajoče enitete, slednje pa se pretvorijo v seznam Asset-ov in vrnejo odjemalcu za prikaz. Podprta je paginacija.
<code>void searchAssets(String searchString, List<String> searchFqdnTypes, int maximumResults);</code>
Sproži splošno iskanje in začasno shrani rezultat iskanja.
<code>PagingLoadResult<Asset> getResults(PagingLoadConfig config);</code>
Iz začasno shranjenih rezultatov splošnega iskanja vrne zahtevano stran.

TreeService

TreeService nudi podporo drevesu kategorij in nekaterim elementom urejevalnika. Vključuje metode za prikaz in preurejanje strukture drevesa ter za dodajanje (oziroma, bolj točno, za povezovanje) vsebin v kategorije. Kategorija je za potrebe prenosa med strežnikom in odjemalcem ter za nadaljnjo uporabo v odjemalcu predstavljena kot instanca razreda Folder.

<code>List<Folder> getFolderChildren(String folder);</code>
Vrne seznam podkategorij za podano kategorijo (argument je primarni ključ kategorije).
<code>Folder addNewFolder(String caption, String parentId);</code>
V podano kategorijo doda novo podkategorijo s podanim imenom in novo ustvarjeno kategorijo vrne.
<code>Folder renameFolder(String caption, String id);</code>

Preimenuje kategorijo in jo vrne.

```
void deleteFolder(String id);
```

Izbriše kategorijo. Rekurzivno pobriše tudi podkategorije.

```
String moveCategory(String categoryId, String destinationCategoryId, int destinationIndex);
```

Premakne podano kategorijo v drugo kategorijo (da postane podkategorija ciljne kategorije).

```
List<Folder> getCategoriesForAsset(String id, String type);
```

Za podano vsebino podanega tipa vrne seznam kategorij, v katere je umeščena oziroma s katerimi je povezana.

```
void removeAssetFromCategory(String assetId, String assetType, String categoryId);
```

Podano vsebino odstrani iz kategorije.

```
void addAssetToCategory(String assetId, String assetType, String categoryId);
```

Poveže podano vsebino s kategorijo, tako da ustvari novo entiteto Relation, in za Relation.title nastavi vrednost atributa, ki se pri ciljnem tipu vsebine uporablja za naslov. Če bi naprimer povezali v kategorijo nek članek, bi veljalo
`Relation.title == Article.title`

```
void addRelationsToCategory(ArrayList<String> relationIds, String destinationCategoryId);
```

Kopira podane povezave in pri kopijah referenco na kategorijo popravi tako, da so vsebine iz teh povezav povezane še s ciljno vsebino.

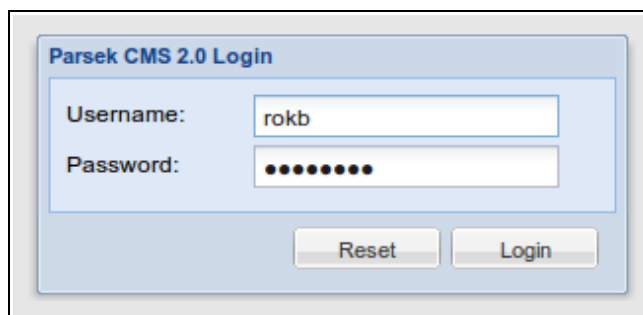
```
void saveCategory(Folder folder);
```

V odjemalcu je na voljo posebno okno s hitrim pregledom lastnosti kategorije in nekatere lastnosti je možno urejati. Po končanju urejanja se pokliče ta metoda, ki shrani spremembe.

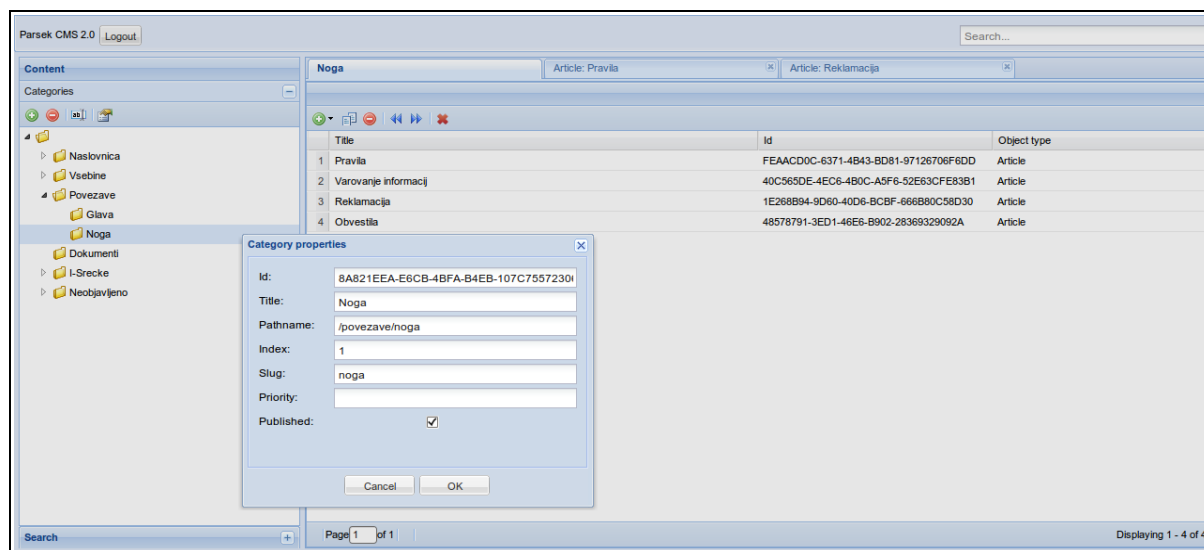
Priloga G: Podrobnosti odjemalca

1. Uporabniški vmesnik

Uporabniški vmesnik je razdeljen na tri dele. Na vrhu je orodna vrstica, ki je zaenkrat še nekoliko neizkoriščena, prikazuje gumb za odjavo, skrajno desno pa vnosno polje za splošno iskanje po vsebinah. Preostali del uporabniškega vmesnika je razdeljen na dva dela. V levem delu lahko izbiramo med drevesom kategorij in naprednim iskanjem, vsebina desnega dela pa je odvisna od izbora v levem delu.



Prijava v sistem



Slika 22: Uporabniški vmesnik

Trenutno je odprto okno s hitrim pregledom lastnosti kategorije. Levo v ozadju je drevo kategorij, desno pa seznam vsebin iz izbrane kategorije. Ob dvokliku na element iz seznama se odpre nov zavihek z urejevalnikom. Trenutno sta za urejanje odprta dva članka.

Na vrhu drevesa kategorij je orodna vrstica z gumbi za dodajanje, brisanje in preimenovanje kategorije, zadnji gumb pa odpre okno s hitrim pregledom lastnosti kategorije. Drevo podpira

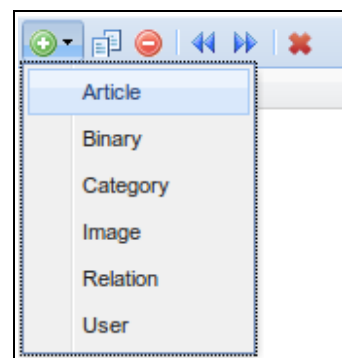
tudi t.i. „drag and drop“, tako da je preurejanje drevesa karseda enostavno. Kategorijo lahko „primemo“ in jo povlečemo na drugo mesto.



Slika 23: Drag and drop

Na levi sliki smo kategorijo *Naslovnica* povlekli med *Glavo* in *Nogo*, kamor bo premaknjena, ko jo izpustimo. Na desni sliki pa smo jo povlekli direktno v kategorijo *Glava*, v katero bo premaknjena, ko jo izpustimo. Kategorij ne moremo premakniti v eno izmed svojih podkategorij, tako kategorije *Povezave* naprimer ne bi mogli povleči v kategorijo *Glava*.

V prvem zavihku je vedno razpredelnica, v kateri se vidi, katere vsebine kategorija vsebuje. Razpredelnica prikazuje naslov vsebine (pravzaprav naslov povezave, torej `Relation.title`). Tudi v tem delu je na vrhu orodna vrstica in sicer z naslednjimi gumbi od leve proti desni:



- Dodajanje nove vsebine – odpre se meni v katerem izberemo tip vsebine. Po izboru se odpre nov zavihek s praznim urejevalnikom za izbran tip vsebine. Vsebina se doda v trenutno izbrano kategorijo v drevesu.
- Kloniranje označeno – klonira izbrano vsebino in jo klon odpre v novem zavihku.
- Odstrani vsebino – ta gumb ne izbriše vsebine iz baze ampak jo samo odstrani iz kategorije (izbriše relacijo, ki povezuje vsebino s to kategorijo)
- Premakni na prejšnji/naslednjo stran – premakne označene vsebine na prejšnjo ali naslednjo stran, če je vsebin preveč in so zato prikazane na več straneh.
- Izbriši vsebino – izbris iz baze, če le ni povezana še s kako drugo kategorijo.

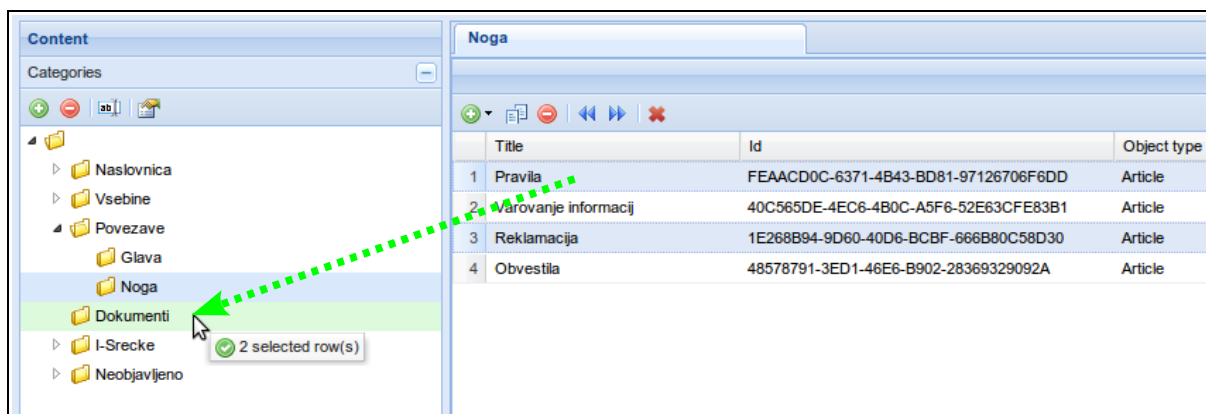
Razen dodajanja nove vsebine, so vse ostale izbire na voljo tudi v kontekstnem meniju s klikom na desni miškin gumb.

V preglednici je prav tako implementiran „drag and drop“ in sicer sta podprti dve funkcionalnosti:

- Izbrane vsebine lahko povlečemo in prenesemo v določeno kategorijo v drevesu kategorij na desni in ustvarile se bodo nove relacije, ki bodo izbrane vsebine povezale

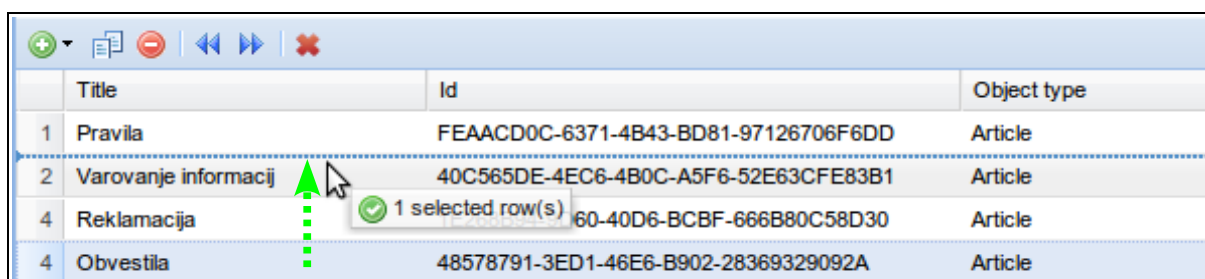
s ciljno kategorijo. Tako lahko vsebine, ki se že nahajajo v neki kategoriji, dodamo še v druge kategorije.

- Izbrane vsebine lahko premeščamo znotraj preglednice in tako urejamo njihov vrstni red; le-ta je namreč lahko pomemben (naprimer pri prikazu vsebin na spletni strani).



Slika 24: Drag and drop – dodajanje vsebin v drugo kategorijo

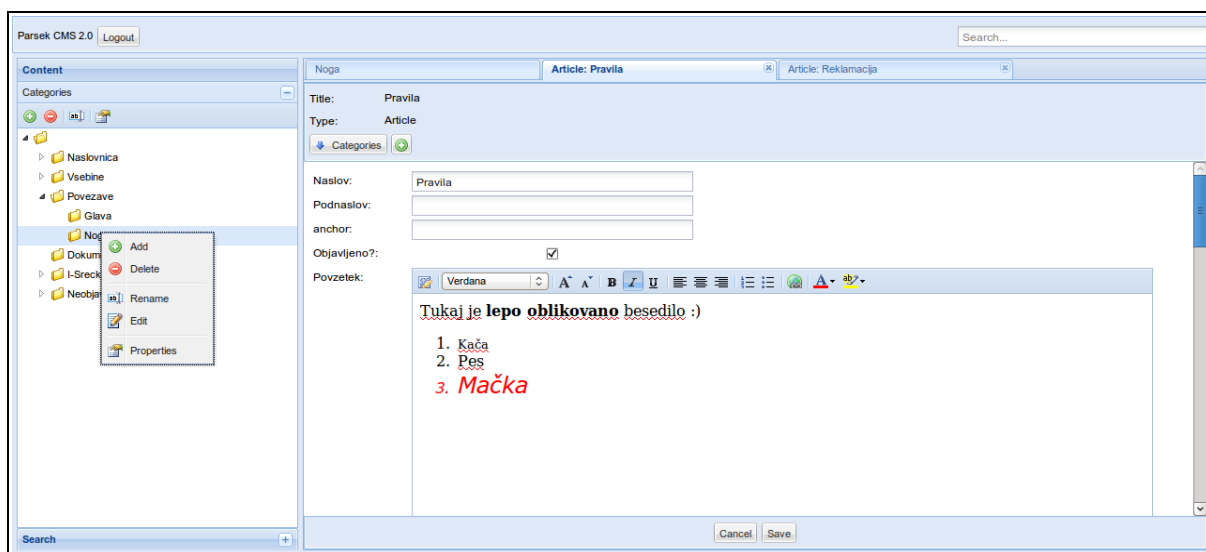
Članka Pravila in Reklamacija iz kategorije Noga, bomo dodali še v kategorijo Dokumenti.



Slika 25: Drag and drop – sprememba vrstnega reda

Članek obvestila bi radi imeli na drugem mestu, takoj za člankom Pravila.

Ob dvokliku na enega od vnosov iz preglednice se zgenerira urejevalnik zanj in prikaže v novem zavihku. Če imamo neko entiteto že odprto v enem od zavihkov, se ne odpre nov zavihek, temveč se samo prikaže že obstoječi. Evidenca odprtih zavihkov se hranijo v obliki razpršene tabele, kjer je ključ sestavljen iz tipa vsebine in primarnega ključa urejane vsebine (npr. „Article-724SF93-9241-A88321“), vrednost pa je referenca na zavihek.



Slika 26: Urejanje članka

Vnosa polja so zgenerirana samodejno, glede na entitetni tip, ki ga urejamo. Na levi je viden tudi meni (to sicer ni povezano z urejevalnikom članka), ki se odpre ob desnem kliku na kategorijo.

2. Generiranje vnosnih polj

2.1 Preprosta vnosna polja

Za preprosta vnosna polja smatramo tista, katerih vsebina niso druge entitete temveč večina osnovnih podatkovnih tipov, ki so v Hibernate ogrodju podprti za attribute entitet. Za njih že obstajajo pripravljeni GUI elementi iz ogrodja GXT, tako da je relativno enostavno poskrbeti za njih.

Pregled enostavnih vnosnih polj:

f.getType()	f.getInputType()	Vnosno polje
STRING	INPUT_TYPE_TEXTBOX	TextField – enovrstično tekstovno vnosno polje
	INPUT_TYPE_TEXTAREA	TextArea – več vrstično tekstovno vnosno polje
	INPUT_TYPE_HTMLEDIT	HtmlEditor – HTML urejevalnik
BOOLEAN,	/	CheckBox – polje s kljukico
INTEGER, LONG, FLOAT	/	NumberField – kot TextField, le da zagotavlja, da je res vnesena številska vrednost.

		Loči med obliko decimalnih in celih števil, itd...
DATE	/	DateField – polje za vnos datuma, vključuje tudi priročen grafičen vmesnik za izbor datuma.
ENUM	/	ComboBox – polje z vnaprej definiranimi možnostmi, med katerimi je možno izbirati.
BINARY	/	FileUploadField – polje za nalaganje datotek.

2.2 Kompleksna vnosna polja

Kompleksna vnosna polja se uporabijo za tiste attribute, ki se nanašajo na neko drugo entiteto oziroma na nivoju podatkovnega modela predstavljajo povezave med tabelami (*one-to-one*, *one-to-many*, itd...). Gradnike uporabniškega vmesnika zanje je bilo treba posebej sprogramirati.

Seznam (*FieldType.LIST*)

Nanj naletimo pri povezavah *one-to-many*, ko je urejana entiteta povezana z več drugimi entitetami. Ko na seznam želimo dodati novo entiteto, se odpre iskalnik entitet, v katerega uporabnik vnese iskalno poizvedbo, iz rezultatov iskanja pa nato izbere ciljno entiteto. Odjemalec za iskanje pokliče oddaljeno metodo `EditorService.searchForAssets`. Izmed rezultatov iskanja nato uporabnik izbere entiteto, ki jo želi dodati.

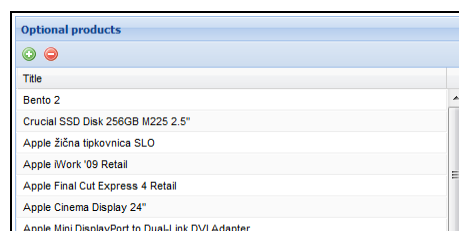


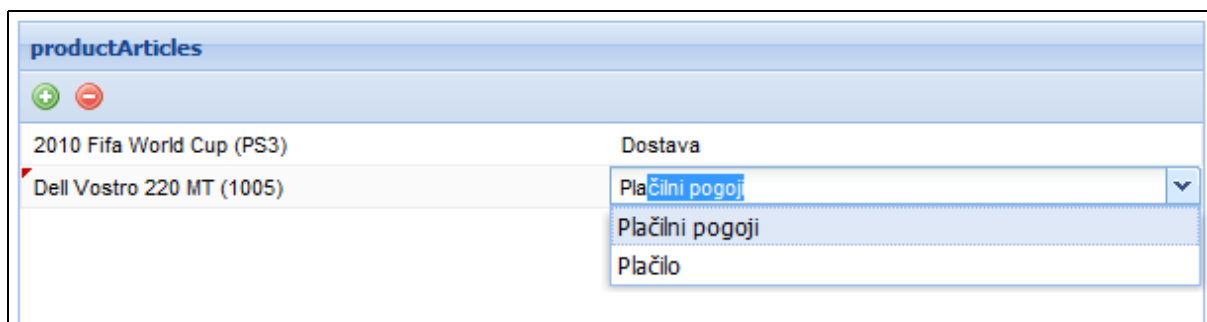
Tabela ključev in vrednosti (*FieldType.MAP*)

Hibernate omogoča v entitetah definirati tudi attribute tipa Map, kjer sta tako ključ in vrednost lahko bodisi enostaven tip (boolean, String, int, Date...), bodisi neka druga entiteta. Imamo torej štiri možne kombinacije, ki so navedene v naslednji preglednici:

Map<tip ključa, tip vrednosti>	Primer
Map<enostaven tip, enostaven tip>	Map<String, String>
Map<enostaven tip, entiteta>	Map<String, OnlinePurchase>
Map<entiteta, enostaven tip>	Map<Customer, Boolean>
Map<entiteta, entiteta>	Map<Property, PropertyValue>

Za potrebe urejanja take podatkovne strukture je bila spisana komponenta MapGrid, ki

prikaže razpredelnico z 2 stolpcema – prvi za ključ, drugi za vrednost. Ob inicializaciji se pripravi ustrezen urejevalnik celice (CellEditor) in izrisovalnik celic (GridCellRenderer) za vsakega od stolpcev, glede na tip, ki ga je potrebno urejati in prikazovati. Urejevalnik se znotraj celice namesto izrisovalnika prikaže ob dvokliku nanjo.



Slika 27: Tabela ključev in vrednosti

Urejanje atributa tipa `Map<Tangible, Article>`. Ob dvokliku na celico se je prikazal padajoči seznam člankov, ki so na voljo. Seznam podpira filtriranje – v zgornjem primeru smo vpisali „pla“ in dobili temu ustrezne možnosti za izbiro.

Prikazovalnik za enostavne tipe v celico preprosto izpiše vrednost (številko, `true/false`, niz znakov, datum, itd...), prikazovalnik za entitete pa izpiše vrednost naslova vsebine `Asset.title`.

Urejevalnik enostavnih tipov znotraj celice prikaže enega izmed enostavni vnosnih polj (opisanih v tabeli v poglavju 4.2.1), za entitete pa prikaže padajoči seznam vseh obstoječih entitet ustreznega entitetnega tipa, izmed katerih nato uporabnik eno izbere.

Padajoči seznam entitet (`FieldType.SUB_ASSET`)

V primeru *one-to-one* in *many-to-one* povezav prikažemo padajoči seznam z entitetami, ki so na voljo. Seznam je enak, kot v urejevalniku celice pri tabeli vrednosti in ključev iz prejšnjega primera oziroma zaslonske slike.

2.3 Prilagojeni urejevalniki

Prilagojeni urejevalniki so namenjeni urejanju tipov vsebin, za katere je bilo ugotovljeno, da bi bilo urejanje učinkovitejše (lažje in hitrejše), če bi bil uporabniški vmesnik nekoliko prilagojen glede na specifične potrebe pri urejanju teh vsebin.

Tako je naprimer pri urejanju podatkov o sliki dobrodošel predogled naložene slike in povezava do nje, če pa gre za navadno datoteko, pa lahko vidimo neposredno povezavo do nje in rezultat MD5 zgoščevalne funkcije.

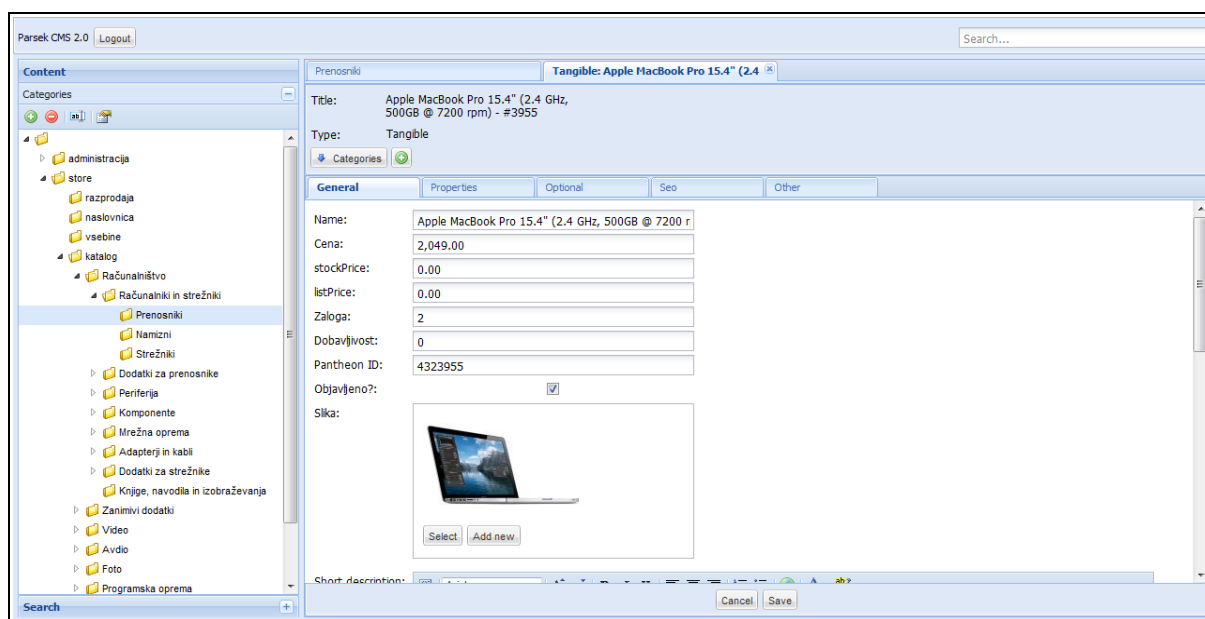
Še posebej pa se je potreba za poseben urejevalnik pokazala pri prodajnih artiklih spletne trgovine, ki imajo veliko atributov, tako da je zgeneriran generični urejevalnik nepraktičen.

Glavna lastnost urejevalnika ProductEditor je, da vnosna polja razporedi med več zavihkov (v trenutni verziji jih je pet), kjer so na prvem tisti, ki se jih najpogostejše ureja ni hkrati nudijo hiter pregled osnovnih informacij o izdelku. Vsak zavihek, razen zadnjega, ima vnaprej določen seznam imen polj, ki naj bi jih vseboval. Zadnji zavihek „Other“ nima svojega seznama temveč avtomatično vsebuje vsa polja, ki s seznamami še niso bila razporejena na enega izmed prejšnjih zavihkov.

Za generiranje polj se pravzaprav še vedno uporabljajo enake metode in algoritmi kot pri generičnem urejevalniku, le da se zgenerirana polja dodajajo na različne FormPanel-e, ki so vsak v svojem zavihku.

```
String[] GENERAL_FIELDS = {"name", "price", "stockPrice", "streetPrice", "listPrice",
    "stock", "availability", "externalID", "published", "image", "shortDescription",
    "description"};
String[] PROPERTIES_FIELDS = {};
String[] OPTIONAL_FIELDS = {"optionalTangibles", "optionalServices"};
String[] SEO_FIELDS = {"title", "anchor", "seoDescription", "keywords"};
```

Seznami polj za zavihke



Slika 28: ProductEditor

Trenutno je prikazan prvi zavihek z osnovnimi podatki.

Poseben zavihek je zavihek „Properties“, ki ima na vrhu padajoči seznam skupin izdelkov s katerim uvrstimo izdelek v eno izmed skupin (naprimer nek določen model računalniškega

monitorja med skupino „Monitorji“). Skupina izdelkov namreč določa skupne lastnosti vseh izdelkov, ki vanjo spadajo (monitorji imajo naprimer diagonalo zaslona, svetilnost, kontrast, itd...). Pod izbirnikom za skupino pa je razpredelnica lastnosti, ki je zelo podobna prej omenjeni tabeli ključev in vrednosti, vendar implementira nekaj specifične logike v zvezi z lastnostmi izdelkov. Vsak izdelek avtomatično podeduje lastnosti iz skupine, lahko pa ima dodatno še svoje lastnosti. Pod ta dva elementa se lahko normalno naniza še druga polja, če so navedena v seznamu (v zgornjem primeru sicer ni nobenega dodatnega polja – seznam PROPERTY_FIELDS je prazen).

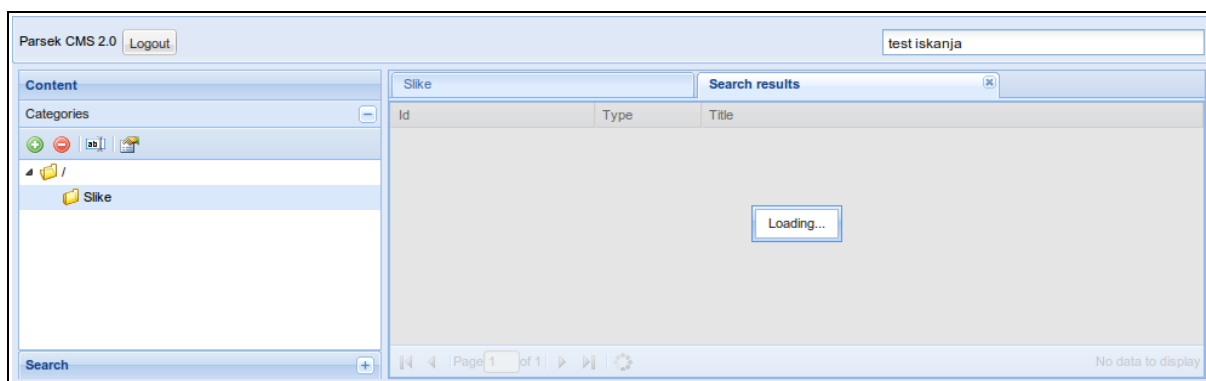
Urejevalnik za slike in datoteke zgoraj desno prikazuje nekatere podatke o naloženi datoteki, ki bi nas utegnili zanimati. Podan je tip vsebine datoteke, velikost datoteke, vrednost md5 zgoščevalne funkcije in direktna povezava na datoteko, preko katere jo lahko z brskalnikom naložimo iz strežnika. V primeru, da gre za sliko, je prikazan še njen predogled (pomanjšana slika) in njene dimenzije.

Slika 29: Urejevalnik slik in datotek

Posebnost tega urejevalnika je prikaz dodatnih informacij o datoteki ter neposredna povezava nanjo. V primeru, da gre za sliko, se prikaže še njen predogled.

3. Iskanje

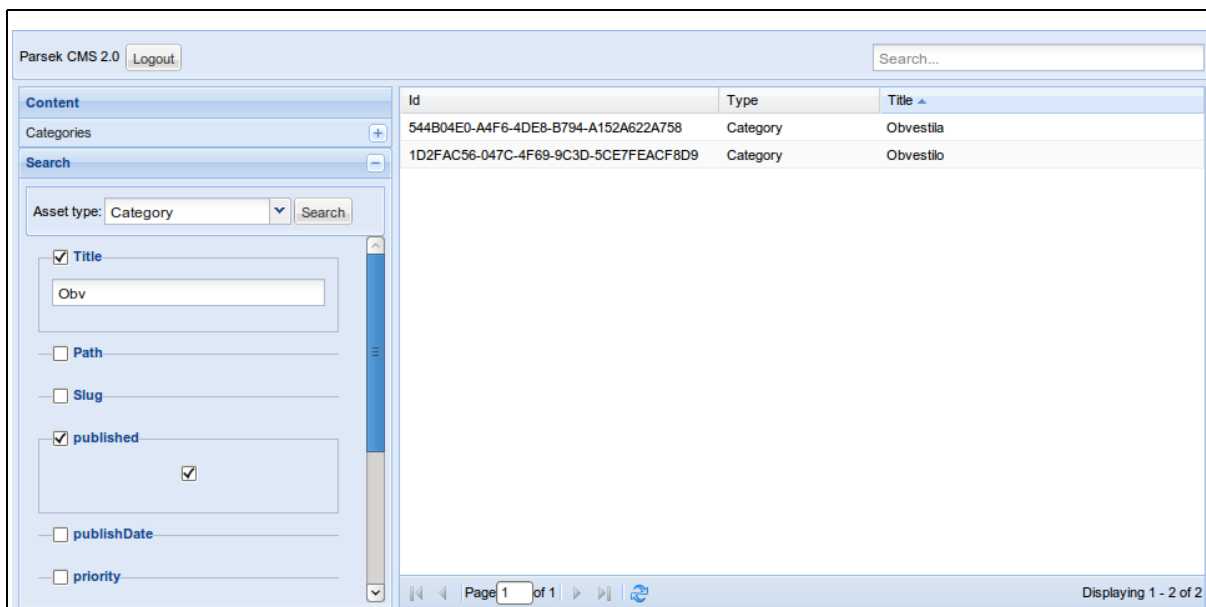
Kot že vemo, obstajata dve vrsti iskanja. Poizvedbo za splošno tekstovno iskanje vnesemo v skrajno zgornje desno polje, ki je vedno vidno na vrhu uporabniškega vmesnika. Ko pritisnemo tipko „Enter“, se izvede iskanje, rezultati pa se prikažejo v posebnem zavihku.



Slika 30: Splošno iskanje

Iskanje po nizu „test iskanja“. Ob pritisku na Enter se je sprožilo iskanje in se prikazal zavihek, v katerem bodo nato prikazani rezultati iskanja.

Za uporabo naprednega iskanja najprej s klikom desno spodaj na vrstico „Search“ razpremo vnosno masko za napredno iskanje. Nato iz padajočega seznama izberemo tip vsebine, po katerem bomo iskali, nazadnje izberemo polja, ki bodo vključena v iskalno poizvedbo ter za izbrana polja vnesemo želene vrednosti. Rezultati iskanja se prikažejo na desni strani.

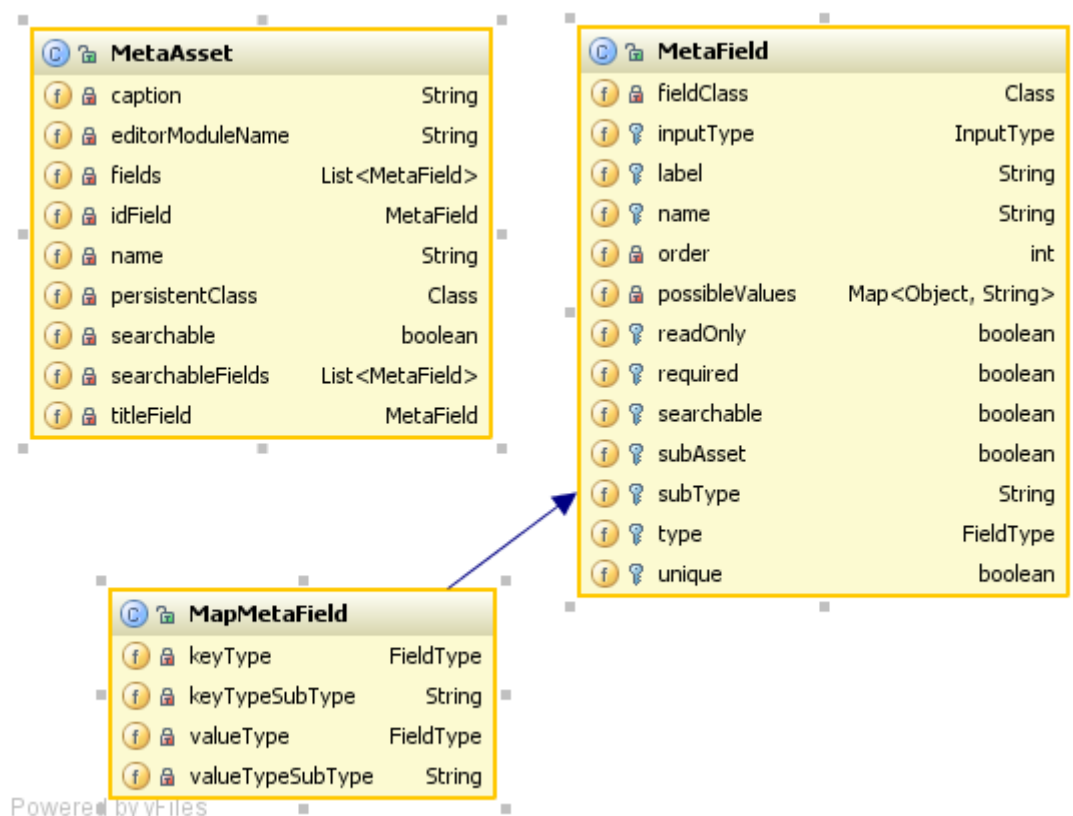


Slika 31: Napredno iskanje

Levo je namesto drevesa prikazana vnosna maska za napredno iskanje. Primer prikazuje iskanje kategorij, ki so označene kot objavljene (published = true) in njihov naslov vsebuje niz „Obv“ (title LIKE '%0bv%'). Ob dvokliku na enega izmed zadetkov iskanja, se le-ta odpre v urejevalniku.

Z naprednim iskanjem lahko najdemo tudi vsebine, ki niso uvrščene v nobeno kategorijo, s čimer je mogoče CMS uporabljati kot neke vrste urejevalnik podatkovne baze tudi kadar drevesna organizacija kategorij še ni izdelana imamo pa v bazi že shranjene podatke. Zelo dobrodošla pa je ta funkcionalnost tudi v primeru, ko ponesreči odstranimo neko vsebino iz vseh kategorij in bi sicer nepovezana „obvisela“ nekje v bazi. Napredni iskalnik lahko s trikom uporabimo tudi za to, da naštejemo vse entitete nekega tipa. To lahko naredimo naprimer za entitetni tip „Category“ tako, da v polje „Title“ vnesemo znak „%“.

Priloga H: Razredni diagrami



Priloga I: Seznam slik

Slika 1: Arhitektura in princip delovanja sistema.....	4
Slika 2: Poenostavljen primer entitet, ki predstavljata kategorijo in povezavo.....	5
Slika 3: Primer drevesne kategorizacije vsebin.....	6
Slika 4: Arhitektura odjemalca.....	7
Slika 5: Skica uporabniškega vmesnika.....	9
Slika 6: Brskanje po drevesu.....	10
Slika 7: Glavne komponente strežnika.....	11
Slika 8: Povezanost komponent.....	13
Slika 9: Drevesni urejevalnik.....	30
Slika 10: Uvedba novega vsebinskega tipa.....	31
Slika 11: Magnolia, Jackrabbit in repozitorij vsebine.....	34
(Vir: [12])	
Slika 12: Hierarhija vsebine spletnega mesta.....	35
(Vir: [14])	
Slika 13: Urejanje vsebine.....	35
(Vir: [14])	
Slika 14: Dialog za urejanje.....	36
(Vir: [14])	
Slika 15: Mehanizem predlog.....	37
(Vir: [12])	
Slika 16: Odnos med vozlišči v JCR in spletno stranjo.....	38
(Vir: [12])	
Slika 17: Odnos med skripto predloge za članek in izrisom članka.....	38
(Vir: [12])	
Slika 18: Cevovod.....	40
Slika 19: Od analize poslovnega objekta do uporabniškega vmesnika.....	41
(Vir: [15])	
Slika 20: Diagram zaporedja pri uporabi GwtMetawidget-a.....	42
(Vir: [15])	
Slika 21: Modificiran diagram zaporedja.....	42
Slika 22: Uporabniški vmesnik.....	70
Slika 23: Drag and drop.....	71
Slika 24: Drag and drop – dodajanje vsebin v drugo kategorijo.....	72
Slika 25: Drag and drop – sprememba vrstnega reda.....	72
Slika 26: Urejanje članka.....	73
Slika 27: Tabela ključev in vrednosti.....	75
Slika 28: ProductEditor.....	76
Slika 29: Urejevalnik slik in datotek.....	77
Slika 30: Splošno iskanje.....	78
Slika 31: Napredno iskanje.....	78

Viri

- [1] (2010) Wikipedia: Model–View–Controller. Dostopno na:
<http://en.wikipedia.org/wiki/Model-view-controller>
- [2] (2010) Seam – Contextual Components. Dostopno na:
<http://docs.jboss.org/seam/latest-2.2/en-US/html/>
- [3] (2010) The Java EE 5 Tutorial. Dostopno na:
<http://download.oracle.com/javaee/5/tutorial/doc/>
- [4] (2010) Hibernate Annotations. Dostopno na:
<http://docs.jboss.org/hibernate/stable/annotations/reference/en/html/>
- [5] (2010) Trail: The Reflection API. Dostopno na:
<http://download.oracle.com/javase/tutorial/reflect/index.html>
- [6] (2008) BeanMap (Commons Collections 3.2.1 API). Dostopno na:
<http://commons.apache.org/collections/apidocs/index.html>
- [7] (2010) Google Web Toolkit Overview Dostopno na:
<http://code.google.com/webtoolkit/overview.html>
- [8] (2010) Google Web Toolkit Javadoc. Dostopno na:
<http://google-web-toolkit.googlecode.com/svn/javadoc/2.1/index.html>
- [9] (2010) Ext GWT 2.2.0 Explorer. Dostopno na:
<http://www.sencha.com/examples/>
- [10] (2010) GXT API 2.2.0. Dostopno na:
<http://dev.sencha.com/deploy/gxtdocs/>
- [11] (2008) Java Bean Support with Ext GWT. Dostopno na:
<http://www.sencha.com/blog/2008/07/14/preview-java-bean-support-with-ext-gwt/>
- [12] (2010) Magnolia Documentation. Dostopno na:
<http://documentation.magnolia-cms.com/>
- [13] (2010) Magnolia Community Wiki. Dostopno na:
<http://wiki.magnolia-cms.com/display/WIKI/Home>
- [14] (2010) Magnolia Live demo – Author instance. Dostopno na:
<http://demoauthor.magnolia-cms.com/>
- [15] (2010) Metawidget Documentation. Dostopno na:
<http://metawidget.sourceforge.net/documentation.html>
- [16] (2010) Metawidget Videos. Dostopno na:
<http://metawidget.sourceforge.net/videos.html>